



universität  
wien

# Programmierung 2

## Vorlesung

Manuel Gall,  
Conrad Indiono,  
Florian Stertz,  
Helmut Wanek (Vortragender),  
Christoph Winter

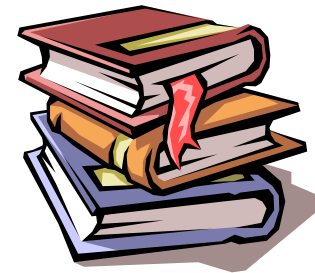
# Literatur

## C++:

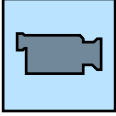
- H.M. Deitel & P.J. Deitel: C++ How to Program, Prentice Hall.
- H.M. Deitel & P.J. Deitel & T.R. Nieto: C++ in the Lab, Prentice Hall (Übungsbuch zu C++ How to Program).
- Scott Meyers: Effective Modern C++, O'Reilly.
- Bjarne Stroustrup: Einführung in die Programmierung mit C++, Pearson Studium.
- Bjarne Stroustrup: Die C++ Programmiersprache. Addison Wesley.
- Bjarne Stroustrup: Programming Principles and Practice Using C++ (2nd Edition), Addison Wesley.
- Herb Sutter: Exceptional C++ Style, Pearson.

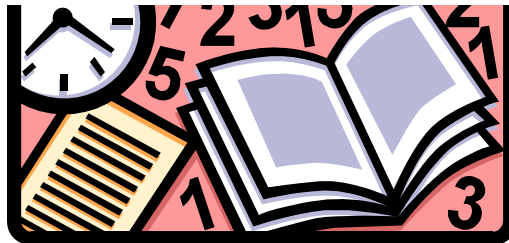
## Java:

- James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, Alex Buckley, Daniel Smith:  
The Java Language Specification, Java SE 8 Edition (Java Series), Oracle.



# Folien des Vortrags

- Online Präsentation im Web
  - erreichbar über die Homepage
  - <https://cewebs.cs.univie.ac.at/PR2/ss19>
  - enthält zusätzlich alle dynamischen Folien (Symbol )





universität  
wien

# 1. Iteratoren

## Klasse Vector (aus PR1)

- Wir setzen im Weiteren die Implementierung der Klasse Vector voraus, die am Ende von PR1 in der Vorlesung beschrieben wurde.
- Sollten Sie keine solche Klasse haben, so müssen Sie eine solche programmieren (dafür kann z.B. die Übungseinheit in dieser Woche verwendet werden).
- Die Klasse wird in den nächsten Wochen erweitert, bis sie einen zu `std::vector` vergleichbaren Funktionsumfang hat.

## Funktionalität der Klasse Vector (1)

- Die Klasse Vector kann beliebig viele double-Werte speichern. Sie wächst bei Bedarf dynamisch und bietet:
  - Konstruktoren:
    - **Defaultkonstruktor** (liefert einen leeren Vektor)
    - **Konstruktor mit Größenangabe** (liefert einen Vektor mit Platz für vorgegebene Anzahl von Elementen)
    - **Konstruktor mit Initialisierungsliste** (liefert einen Vektor mit spezifiziertem Inhalt)
    - **Kopierkonstruktor** (liefert einen Vektor mit demselben Inhalt)
  - Destruktor
  - Kopierzuweisungoperator
  - Ausgabeoperator
  - Methoden: ...
  -

## Funktionalität der Klasse Vector (2)

- Methoden:

- `size()` (liefert die Anzahl der gespeicherten Elemente)
- `empty()` (liefert `true`, falls der Vektor leer ist, `false` sonst)
- `clear()` (löscht alle Elemente aus dem Vektor)
- `reserve()` (Kapazität des Vektors wird vergrößert, falls nötig)
- `shrink_to_fit()` (Kapazität des Vektors wird auf Anzahl der Elemente reduziert)
- `push_back()` (fügt ein neues Element am Ende hinzu)
- `pop_back()` (entfernt ein Element vom Ende)
- `capacity()` (liefert die Größe des reservierten Speichers als Anzahl Elemente)
- `operator[]` (`const` und nicht `const`; abgesichert;  
retourniert Element an gegebener Position)

- Anmerkung: In PR1 wurden zu Übungszwecken auch noch eine Reihe anderer Methoden und globaler Funktionen beschrieben, die für uns aber im Weiteren keine Rolle mehr spielen werden.

## Vergleich mit `std::vector`

- Neben kleinen Unterschieden in der Funktionalität (z.B. abgesicherter Operator `[]`) und fehlenden Methoden (z.B. `at`) sind vor allem zwei große Unterschiede auszumachen:
- Vector kann nur `double` Werte speichern, `std::vector` "beliebige" Werte.  
(→Templates; wird später realisiert)
- Vector bietet keine Iteratoren an, `std::vector` schon.  
Vector kann daher nicht in range-based-for-loop verwendet werden oder zusammen mit den Algorithmen der STL.



## Iterator (Wiederholung)

- Iteratoren sind eine Verallgemeinerung des Pointerkonzepts (oder des Referenzkonzepts). Ein Iterator referenziert einen Wert in einem Container. Es werden zumindest die beiden Operatoren `*` (Dereferenzieren) und `++` (Prefix Inkrement; Weiterschalten zum nächsten Wert) angeboten.
- Weitere Operationen sind optional. Im Kontext mit range-based-for-loops ist `!=` (Vergleich zweier Iteratoren) notwendig (meist wird dann auch `==` angeboten).
- Der C++ -Standard definiert einige Iterator-Typen, die unterschiedliche Operationen anbieten. Es ist genau festgelegt, welche Operationen ein bestimmter Typ anbieten muss. Die Details sind komplex.

# Iterator Arten

Iterator category					Defined operations
<a href="#">ContiguousIterator</a>	<a href="#">RandomAccessIterator</a>	<a href="#">BidirectionalIterator</a>	<a href="#">ForwardIterator</a>	<a href="#">InputIterator</a>	<ul style="list-style-type: none"><li>•read</li><li>•increment (without multiple passes)</li></ul>
					<ul style="list-style-type: none"><li>•increment (with multiple passes)</li></ul>
					<ul style="list-style-type: none"><li>•decrement</li></ul>
					<ul style="list-style-type: none"><li>•random access</li></ul>
					<ul style="list-style-type: none"><li>•contiguous storage</li></ul>
Iterators that fall into one of the above categories and also meet the requirements of <a href="#">OutputIterator</a> are called mutable iterators.					
<a href="#">OutputIterator</a>					<ul style="list-style-type: none"><li>•write</li><li>•increment (without multiple passes)</li></ul>

## Iterator Zustände

- Obwohl die Operationen Dereferenzieren und Inkrement immer angeboten werden müssen, kann deren Anwendung auf einen Iterator in bestimmten Zuständen illegal sein. Wir unterscheiden daher folgende Zustände:
  - Dereferenzierbar (dereferenceable): Der Iterator kann dereferenziert werden und liefert einen Wert. Der von `end()` gelieferte Iterator ist nicht dereferenzierbar, ebenso wie eventuell ein Input- oder Output-Iterator, der bereits dereferenziert wurde (single-pass).
  - Inkrementierbar (incrementable): Der Iterator kann zum nächsten Wert weitergeschaltet werden. Der von `end()` gelieferte Iterator ist nicht inkrementierbar. Ein Output-Iterator könnte beispielsweise immer abwechselnd dereferenzierbar und inkrementierbar sein.
  - Anwenden von dem Zustand nicht entsprechenden Operationen führt zu undefiniertem Verhalten!
-

# Invalidierung (invalidation) von Iteratoren

- Durch Operationen am Container wie Löschen oder Einfügen können Iteratoren ungültig (invalidiert) werden. Invalidierte Iteratoren sind weder dereferenzierbar, noch inkrementierbar.
- Welche Operationen bei welchen Containern zur Invalidierung von (bestimmten) Iteratoren führen, ist in der Dokumentation der jeweiligen Operationen zu finden.
- Im Anhang findet sich eine kurze [Übersicht](#).

## Bereich (Range)

- Durch Angabe zweier Iteratoren kann ein Bereich festgelegt werden. Ein Bereich ist nur gültig, wenn beide Iteratoren valid sind und durch fortgesetztes Inkrementieren des ersten Iterators irgendwann einmal der zweite Iterator erreicht wird.
- Ungültige Bereiche führen bei Verwendung zu undefiniertem Verhalten!
- Die Methoden `begin()` und `end()` liefern bei STL-Containern einen Bereich, der alle Elemente des Containers umfasst. (`begin()` liefert einen Iterator, der das erste Element referenziert, `end()` einen Iterator, der das – virtuelle – Element nach dem letzten Element referenziert).

## Range-Based-For-Loop

- Bietet ein Container Iteratoren mit zumindest den Operationen ++ (Prefix), \* und != an, sowie die Methoden begin() und end() an, dann kann dieser Container mittels einer range-based-for-loop iteriert werden.

```
for (const auto& elem : container) ...
```

# Algorithmen

- Vordefinierte Algorithmen sind Funktionstemplates, die häufig benötigte Funktionalität für Container zur Verfügung stellen. Sie haben in der Regel einen Bereich, oder Iteratoren als Parameter.
- Suchen, Sortieren, Mischen, Minimum, Maximum, ...
- Wir verzichten auf eine vollständige Aufzählung, werden aber immer wieder einzelne Algorithmen für verschiedene Aufgabenstellungen verwenden.
- Manche Algorithmen verlangen spezielle Typen von Iteratoren (z.B. bidirektional oder multi-pass).

## Iteratoren für unsere Vector Klasse

- Da die Daten in unserer Klasse zusammenhängend (contiguous) gespeichert sind, reichen analog zu `std::vector` einfache Pointer als Iteratoren aus.
- Wir müssen also nur die Methoden `begin()` und `end()` (jeweils für den konstanten und nicht konstanten Fall) implementieren:

```
double* begin() {return values;}  
double* end() {return values+sz;}  
const double* begin() const {return values;}  
const double* end() const {return values+sz;}
```

- Unsere Klasse `Vector` ist damit bereits tauglich für range-based-for und STL-Algorithmen.



# Steigerung des Komforts

- Um BenutzerInnen der Klasse, die von den internen Gegebenheiten nichts wissen, die Verwendung der Klasse zu erleichtern, sollten einige Datentypen deklariert werden (wie z.B. der Typ der Iteratoren etc.). Der Standard empfiehlt:

value_type	Typ der gespeicherten Elemente
allocator_type	(in unserer LV nicht verwendet)
size_type	Typ von Größenangaben (üblicherweise std::size_t)
difference_type	Typ des Abstands zwischen zwei Elementen/Iteratoren (üblicherweise std::ptrdiff_t; #include <cstddef>)
reference	value_type&
const_reference	const value_type&
pointer	value_type* (im Standard komplizierter, für uns ausreichend)
const_pointer	const value_type* (im Standard komplizierter, für uns ausreichend)
iterator	Typ des Iterators
const_iterator	Typ des const-Iterators
reverse_iterator	(bis auf Weiteres in unserer LV nicht verwendet)
const_reverse_iterator	(bis auf Weiteres in unserer LV nicht verwendet)

# Typdeklarationen mit using alias

- In der Definition der Klasse Vector (im Header-File):

```
public:  
    using value_type = double;  
    using size_type = size_t;  
    using difference_type = ptrdiff_t;  
    using reference = double&;  
    using const_reference = const double&;  
    using pointer = double*;  
    using const_pointer = const double*;  
    using iterator = double*;  
    using const_iterator = const double*;
```

- Wir haben using bereits als using-Deklaration (using std::cin) oder using-Direktive (using namespace std) kennengelernt. Alias für Typen ist eine neue Verwendung und entspricht dem früher verwendeten typedef. using ist speziell für Templates entworfen worden und kann auch alles, was typedef kann. Die Syntax ist außerdem einfacher lesbar (z.B.: `typedef double& reference`).
- Es wird kein neuer Datentyp definiert, sondern nur einem bereits bekannten Typ ein neuer Name (alias) gegeben.

# Verwendung der Typdeklarationen

- Z.B.:

```
for (Vector::const_reference elem : container) ...
```

# Eine Iterator Klasse

- Wir werden uns eine eigene Iterator Klasse definieren. Das gibt uns die Möglichkeit z.B. gegen Dereferenzierung eines nicht dereferenzierbaren Iterators abzusichern und die Reihenfolge der Abarbeitung der Elemente beliebig festzulegen, etwa in umgekehrter Richtung.
- Wir werden unsere Iterator Klasse als verschachtelte (nested) Klasse definieren.

```
class Vector {  
public:  
    class Iterator { //automatisch friend von Vector  
        double *ptr;  
    public:  
        Iterator(double*);  
        Iterator& operator++();  
        bool operator!=(const Iterator&);  
        double& operator*();  
        const double& operator*() const;  
    };  
    ...  
    //Aliase für iterator und const_iterator, sowie begin() und end() Methoden  
    //anpassen
```

# Übungsaufgaben

- Iterator-Klasse so ändern, dass Elemente in umgekehrter Reihenfolge durchlaufen werden. (Eine Änderung von `begin()` und `end()` Methoden wird auch nötig sein.)
- Iterator-Klasse so ändern, dass Elemente zwei mal durchlaufen werden. (Also z.B. 1,2,3,1,2,3 für einen Vektor, der die Elemente 1,2,3 enthält.)
- Iterator-Klasse so ändern, dass jedes Element drei mal durchlaufen wird. (Also z.B. 1,1,1,2,2,2,3,3,3 für den Vektor, der die Elemente 1,2,3 enthält.)



## 2. Die Vektor-Methoden erase und insert

## Erase und Insert

- Nachdem wir die Klasse Vektor mit Iteratoren versehen haben, können wir die Methoden `erase` und `insert` analog zu `std::vector` realisieren. (Wir werden nur die simplen Versionen betrachten, wo genau ein Wert eingefügt, bzw. gelöscht wird.)
- Beide Methoden erhalten einen Iterator (seit C++11 einen `const_iterator`), der die Position angibt. Dabei löscht `erase` genau den Wert, auf den der Iterator verweist und `insert` fügt vor dem referenzierten Wert ein.
- Beide Methoden invalidieren alle Iteratoren (sowie Pointer und Referenzen), die auf Werte nach der Einfüge- bzw. Löschposition verweisen. Beim Einfügen können sogar alle Iteratoren (sowie Pointer und Referenzen) invalidiert werden, wenn es zu einer Allokation eines größeren Speicherbereichs kommt.
- Beide Methoden retournieren einen Iterator, der auf das eingefügte Element oder auf das Element, das auf das gelöschte Element folgt, verweist.

# Achtung

- Das Nichtbeachten der Invalidierung von Iteratoren durch erase bzw. insert kann zu schweren Fehlern führen!

```
for (auto& elem : v) {  
    ...  
    erase bzw. insert Aufruf für v  
    ...  
}
```

```
for (auto it = v.begin(); it != v.end(); ++it) {  
    if (*it == 0) erase(it);  
}
```

```
for (auto it = v.begin(); it != v.end();) {  
    if (*it == 0) it = erase(it);  
    else ++it;  
}
```



# implizite Typumwandlung von iterator nach const\_iterator

- Da der Positionsparameter von erase und insert seit C++11 ein const\_iterator ist und in der Regel iterator und const\_iterator zwei unterschiedliche Klassen (siehe nächste Übungseinheit) sind, benötigen wir eine implizite Typumwandlung von iterator nach const\_iterator. Damit können wir erase und insert dann auch mit Positionsparametern vom Typ iterator aufrufen.
  - Zwei Möglichkeiten für so eine Typumwandlung haben wir kennen gelernt:
    1. Konstruktor in Klasse const\_iterator mit einem Parameter vom Typ const iterator& (const und & optional).  
Schwierig, da const\_iterator Zugriff auf den internen Zustand von iterator bräuchte.
    2. operator ConstIterator() Methode, die einen passenden const\_iterator retourniert (ConstIterator ist dabei der Name der Klasse für const\_iterator).  
Zu bevorzugen, da einfach nur der Konstruktor von ConstIterator mit den passenden Werten aufgerufen werden muss.
-

## operator-

- Für die Implementierung von `erase` bzw. `insert` ist es notwendig, herauszufinden, auf welche Position im Feld sich der Iterator, der als Parameter übergeben wurde, bezieht.
- Um Probleme mit der Datenkapselung zu vermeiden, verwenden wir `operator-`, um die Differenz zwischen zwei Iteratoren zu berechnen (= Anzahl der Inkrementierungen / Dekrementierungen, um vom Start zum Ende zu kommen).
- Dieser Operator ist einfach zu implementieren und wird für mächtigere Iteratortypen (z.B. `RandomAccessIterator`) ohnehin benötigt.
- Da `operator-` für Iteratoren in unserem Vektor genau der Differenz der zugehörigen Pointer bzw. Indizes in das Array entspricht, ergibt sich eine direkte Übersetzung von Iterator zu Pointer bzw. Index.

## operator- Implementierung

- Um Probleme mit Parameterkombinationen aus `iterator` und `const_iterator` zu vermeiden, empfiehlt es sich, `operator-` nicht als Methode (member-Funktion) sondern als globale friend-Funktion zu implementieren. Es reicht dann eine Funktion mit zwei Parametern vom Typ `const_iterator`. `iterator` kann, wegen der impliziten Typumwandlung dann ebenfalls als Parametertyp verwendet werden.
- Eine mögliche Implementierung (unter der Annahme, dass `const_iterator` die Klasse `ConstIterator` ist und die Instanzvariable `ptr` ein Pointer auf das referenzierte Element im Array ist):

```
friend Vector::difference_type operator-(const Vector::ConstIterator& lop,
                                         const Vector::ConstIterator& rop) {
    return lop.ptr-rop.ptr;
}
```

- Analoge Überlegungen gelten auch für `operator==` und `operator!=`

## erase

- Eine mögliche Implementierung unter Verwendung von operator- für Iteratoren

```
Vector::iterator Vector::erase(Vector::const_iterator pos) {  
    auto diff = pos-begin();  
  
    if (diff<0 || static_cast<size_type>(diff)>=sz)  
        throw runtime_error("Iterator out of bounds");  
  
    size_type current{static_cast<size_type>(diff)};  
    for (size_type i{current}; i<sz-1; ++i)  
        values[i]=values[i+1];  
  
    --sz;  
  
    return Vector::iterator{values+current};  
}
```

# insert

- Eine mögliche Implementierung unter Verwendung von operator- für Iteratoren

```
Vector::iterator Vector::insert(Vector::const_iterator pos,
                                Vector::const_reference val) {
    auto diff = pos-begin();
    if (diff<0 || static_cast<size_type>(diff)>sz)
        throw runtime_error("Iterator out of bounds");

    size_type current{static_cast<size_type>(diff)};
    if (sz>=max_sz)
        reserve(max_sz*2); //max_sz*2+10, wenn Ihr Container max_sz==0 erlaubt

    for (size_type i {sz}; i-->current;)
        values[i+1]=values[i];
    values[current]=val;

    ++sz;

    return Vector::iterator{values+current};
}
```

---



universität  
wien

## 3. Algorithmen

# Algorithmen (1)

- Es gibt eine große Anzahl von Algorithmen, die für STL-Container definiert sind. Wenn man vor der Aufgabe steht, eine Ansammlung von Werten in einer gewissen Art und Weise zu bearbeiten, lohnt sich fast immer ein Blick in die Liste dieser Algorithmen. Meist ist das Problem schon (korrekt und effizient) gelöst.
- Wir werden hier nur ein paar Algorithmen exemplarisch betrachten.
- Die meisten Algorithmen werden in der Library `algorithm` (`#include algorithm`) definiert. Auf Ausnahmen wird speziell hingewiesen.
- Die meisten Algorithmen bieten ab C++17 eine Version mit einem zusätzlichen, ersten Parameter an, mit dem die Execution Policy (sequentiell, parallel, `parallel_unsequenced`) gewählt werden kann. Darauf werden wir hier nicht näher eingehen.

## Algorithmen (2)

- Die Anforderungen an die Iteratoren, die als Parameter an Algorithmen übergeben werden, sind je nach Algorithmus unterschiedlich. So verlangt beispielsweise `sort` Random-Access-Iteratoren, während `max_element` mit Forward-Iteratoren auskommt. Verwendung von Iteratoren mit nicht ausreichender Funktionalität führt zu (manchmal schwer lesbaren) Fehlermeldungen.
- Alle Funktionen sind im Namensraum `std` definiert. Wir nehmen an, dass `using namespace std` verwendet wird und verzichten im Weiteren auf die Qualifikation mit `std::`.



## max\_element, min\_element

- `ForwardIt max_element(ForwardIt first, ForwardIt last);`
  - Verwendet operator<. Falls Maximum öfters auftritt, wird die erste Position geliefert.
- `ForwardIt max_element(ForwardIt first, ForwardIt last, Compare cmp);`
  - Verwendet die Funktion cmp, um Elemente zu vergleichen. Diese hat zwei Parameter und retourniert true, wenn der erste Parameter kleiner als der zweite ist

```
std::vector<Person> v;  
...  
//suche die aelteste Person in v  
auto it = max_element(v.begin(), v.end(),  
    [] (const Person& p1, const Person& p2)  
    {return p1.getAlter()<p2.getAlter();}  
    );
```

- `ForwardIt min_element(ForwardIt first, ForwardIt last);`
- `ForwardIt min_element(ForwardIt first, ForwardIt last, Compare cmp);`
  - analog

# accumulate

- Definiert in Library numeric (`#include<numeric>`)

```
T accumulate( InputIt first, InputIt last, T init );
```

- Berechnet die Summe der Werte (init ist der Startwert)

```
T accumulate( InputIt first, InputIt last, T init, BinaryOperation op );
```

- op definiert die Akkumulationsfunktion. Der erste Parameter ist das Zwischenergebnis (Typ T), der zweite das aktuelle Element.

```
std::vector<Angestellte> v;  
...  
//bestimme das Gesamtgehalt aller Angestellten  
double gesamt = accumulate(v.begin(), v.end(), 0.,  
                           [](const double& sum, const Person& p)  
                           {return sum + p.getGehalt();}  
                           );
```

- Akkumulation muss nicht immer Addition sein.

## is\_sorted, sort, nth\_element

```
bool is_sorted( ForwardIt first, ForwardIt last );
```

```
bool is_sorted( ForwardIt first, ForwardIt last, Compare comp );
```

- Liefern true, wenn der Bereich bezüglich operator< bzw. comp sortiert ist.

```
void sort( RandomIt first, RandomIt last );
```

```
void sort( RandomIt first, RandomIt last, Compare comp );
```

- Sortiert den Bereich bezüglich operator< bzw. comp. Random-Iterator erforderlich

```
void nth_element( RandomIt first, RandomIt nth, RandomIt last );
```

```
void nth_element( RandomIt first, RandomIt nth, RandomIt last, Compare comp );
```

- Ordnet so um, dass an der Stelle nth das korrekt einsortierte Element (bezüglich operator< bzw. comp) steht und davor nur Elemente, die kleiner gleich sind. Effizienter, als alles zu sortieren.

## count, count\_if, find, find\_if, find\_if\_not

```
difference_type count( InputIt first, InputIt last, const T &value );
```

```
difference_type count_if( InputIt first, InputIt last, UnaryPredicate p );
```

- Liefern die Anzahl von Elementen im Bereich, die gleich value (bezüglich operator==) sind bzw. für die p true liefert. p ist eine Funktion mit einem Parameter vom Typ const\_reference und liefert einen booleschen Wert.

```
InputIt find( InputIt first, InputIt last, const T& value );
```

```
InputIt find_if( InputIt first, InputIt last, UnaryPredicate p );
```

```
InputIt find_if_not( InputIt first, InputIt last, UnaryPredicate q );
```

- Liefern die erste Position, an der das Element gleich value (operator==) ist oder p true bzw. false liefert.

## all\_of, any\_of, none\_of, for\_each

```
bool all_of( InputIt first, InputIt last, UnaryPredicate p );  
bool any_of( InputIt first, InputIt last, UnaryPredicate p );  
bool none_of( InputIt first, InputIt last, UnaryPredicate p );
```

- Liefern true, wenn p für alle, mindestens eines oder gar keines der Elemente im Bereich true liefert.

```
UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f );
```

- Ruft f für jedes Element im Bereich einmal auf. f ist eine Funktion, die einen Parameter vom Typ const\_reference bzw. reference hat und keinen Returnwert (void). Falls der Parameter den Typ reference hat, kann f die Elemente im Bereich verändern. Der Returnwert von for\_each war bis C++11 die Funktion f, seit C++11 wird std::move(f) retourniert. Ähnlich wie range-based-for-loop.

## copy, copy\_if, transform

```
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );
```

```
OutputIt copy_if( InputIt first, InputIt last, OutputIt d_first,  
                  UnaryPredicate pred );
```

- Kopiere alle Elemente bzw. die Elemente für die `pred true` liefert aus dem Bereich in einen anderen Bereich, der bei `d_first` startet. Retourniert wird ein Iterator auf das Element nach dem zuletzt eingefügten (eventuell `end()`). Falls die beiden Bereiche überlappen und `d_first` liegt zwischen `first` und `last`, dann ist das Verhalten undefiniert. In diesem Fall kann `copy_backward` verwendet werden.

```
OutputIt transform( InputIt first1, InputIt last1, OutputIt d_first,  
                    UnaryOperation unary_op );
```

```
OutputIt transform( InputIt1 first1, InputIt1 last1, InputIt2 first2,  
                    OutputIt d_first, BinaryOperation binary_op );
```

- Speichern das Ergebnis der Operation `unary_op` für jedes Element aus Bereich1 bzw. von `binary op` für jeweils ein Element aus Bereich1 und aus Bereich2 in den Ergebnisbereich der mit `d_first` beginnt. Retourniert wird ein Iterator nach dem zuletzt gespeicherten Ergebnis.
- Es ist darauf zu achten, dass die Bereiche in die kopiert bzw. geschrieben wird, groß genug sind. Sonst erhält man undefiniertes Verhalten. Will man Werte zu einem Bereich hinzufügen, kann man `std::back_inserter` bzw. `std::front_inserter` als Iterator für den Beginn des Zielbereichs verwenden.

## remove, remove\_if

```
ForwardIt remove( ForwardIt first, ForwardIt last, const T& value );
```

```
ForwardIt remove_if( ForwardIt first, ForwardIt last, UnaryPredicate p );
```

- Löschen alle Elemente aus dem Bereich, die gleich (operator==) value sind bzw. für die p den Wert true liefert. Retourniert einen Iterator hinter dem letzten Element des neuen Bereichs. Die Elemente werden nur überschrieben (durch Elemente, die nicht entfernt werden sollen und entsprechend kopiert werden) und nicht wirklich gelöscht. Daher folgt in der Regel anschließend ein Aufruf der Container-Methode erase:
  - `v.erase(remove(v.begin(), v.end(), 0), v.end());`



## 4. Vektor für „beliebige“ Klassen



# Vektor Verallgemeinerung

- In unserer Vektor-Klasse können aktuell nur Werte vom Datentyp double gespeichert werden.
- Durch die Verwendung von Templates kann Vektor so erweitert werden, dass Objekte von nahezu beliebigem Datentyp gespeichert werden können.

# Vorgehensweise

1. Klasse Vektor wird zu einem Template mit einem Typparameter

```
template <typename T>  
class Vector {...};
```

2. Ersetzen des Datentyps double als Elementtyp durch T (weniger Arbeit, wenn man schon fein säuberlich die Typ-Aliase verwendet hat, ansonsten jetzt eine gute Möglichkeit, das nachzuholen).
3. Die Definitionen der Template-Methoden kommen ebenfalls in die Header-Datei (vector.h). Diese Definitionen werden vom Compiler für die Instanziierung bei Bedarf benötigt.
4. Eventuelle Fehler beheben und Testen mit unterschiedlichen Datentypen.



universität  
wien

## 5. STL Container

# STL Container Typen

- Die STL bietet 13 unterschiedliche Container und 3 Container-Adapter. Die Container sind in drei Typen unterteilt:
- Sequenz Container (sequence containers): Für den sequentiellen Zugriff auf die gespeicherten Daten.
- Assoziative Container (associative containers): Sortiert, für die schnelle Suche nach gespeicherten Daten.
- Ungeordnete Assoziative Container (unordered associative containers): Unsortiert, Verwendung von Hashfunktionen, um besonders schnelle Suche nach gespeicherten Daten zu ermöglichen.

# STL Container Beziehungen

- Jeder STL Container ist eine eigene (Template-)Klasse. Es gibt keine gemeinsame Basisklasse (auch nicht innerhalb eines Container-Typs).
- Man geht davon aus, dass die Container so unterschiedlich sind, dass eine Entscheidung für einen bestimmten Container ohnehin notwendig ist und später nicht revidiert werden muss.

# Sequenz Container

- Bieten die Möglichkeit, die gespeicherten Daten sequentiell zu durchlaufen (in der Reihenfolge, wie sie gespeichert wurden).
  - `array` (`#include<array>`): Abstraktion von C++ Arrays mit fixer Größe und abgesichertem Zugriff mittels `at()`.
  - `vector` (`#include<vector>`): Container, der automatisch mit der Anzahl der gespeicherten Werte wächst. Wachstum ist wirklich "effizient" nur, wenn am Ende eingefügt wird.
  - `deque` (`#include<deque>`): Double ended queue: Container, der automatisch mit der Anzahl der gespeicherten Werte wächst. Einfügen und Löschen ist am Anfang und am Ende "effizient" möglich.
  - `forward_list` (`#include<forward_list>`): Einfach verkettete Liste: Einfügen und Löschen überall effizient möglich. Zugriff auf ein Element über seine Position ist aber aufwendig, da die Liste durchlaufen werden muss.
  - `list` (`#include<list>`): Doppelt verkettete Liste: wie `forward_list`, nur dass die gespeicherten Werte in beiden Richtungen durchlaufen werden können.
-

# Assoziative Container

- Halten die gespeicherten Daten in sortierter Reihenfolge und unterstützen daher schnelle Suche, sowie effizientes Einfügen und Löschen (Schnelle Suche alleine könnte z.B. auch mit einem sortierten vector erreicht werden).
- Unterschied zwischen Äquivalenz ( $!(a < b) \ \&\& \ !(b < a)$ ) und Gleichheit (`operator==`), kann zu unerwünschten Effekten führen, wenn die Operatoren nicht entsprechend definiert sind.
- `set` (`#include<set>`): Abstraktion einer Menge. Jeder Wert kann nur einmal gespeichert werden.
- `multiset` (`#include<set>`): Werte können öfters gespeichert werden.
- `map` (`#include<map>`): Ähnlich wie `set`, allerdings werden Paare von Werten mit eindeutigen, unveränderlichen Schlüssel und zugehörigen veränderlichen Werten gespeichert.
- `multimap` (`#include<map>`): Wie `map`, aber Schlüssel können öfters vorkommen.
- Seit C++11 sind `set::iterator` und `multiset::iterator` `const_iterator`en. Gespeicherte Werte können also mittels Dereferenzieren eines Iterators nicht verändert werden.

# Ungeordnete Assoziative Container

- Im Wesentlichen Hashtabellen. Default Hashfunktionen sind für die primitiven C++-Datentypen und einige Klassen wie z.B. string vordefiniert. Für die eigenen Klassen muss eine eigene Hashfunktion definiert werden.
- Sehr effizienter Zugriff (Suche), Einfügen und Löschen ebenfalls effizient. Um die Performance nicht zu stören, darf der Container aber in der Regel nicht ganz gefüllt werden.
- `unordered_set` (`#include<unordered_set>`): wie set, nur Hashtabelle.
- `unordered_multiset` (`#include<unordered_set>`): wie multiset, nur Hashtabelle.
- `unordered_map` (`#include<unordered_map>`): wie map, nur Hashtabelle.
- `unordered_multimap` (`#include<unordered_map>`): wie multimap, nur Hashtabelle



# Container Adapter

- Implementieren ein anderes Interface über eine der Container Klassen.
- `stack (#include<stack>)`: stack Interface (LIFO mit push und pop) über vector, deque oder list.
- `queue (#include<queue>)`: queue Interface (Warteschlange mit push und pop – statt enqueue und dequeue) über deque oder list.
- `priority_queue (#include<queue>)`: priority queue (max heap mit push und pop) Interface über vector oder deque

# Übersicht der STL Container

- <http://en.cppreference.com/w/cpp/container>

# Verwendung von Containern

- Die Auswahl des "richtigen" Containers für eine bestimmte Aufgabenstellung ist nicht trivial. Man sollte die benötigten Operationen (Suchen, Einfügen, Ändern, Sortierte Reihenfolge, Bereichsabfragen, ...) sowie deren Häufigkeit in Betracht ziehen und dann den am besten geeigneten Container wählen (eventuell muss neben der Laufzeiteffizienz auch die Speicherplatzeffizienz mit betrachtet werden).
  - Wenn für die Implementierung einer gewünschten Funktionalität mehrere Optionen zur Verfügung stehen, sollte man folgendermaßen priorisieren:
    1. Verwendung von Methoden des Containers (diese sind besonders gut für die genaue Implementierung des Containers optimiert).
    2. Verwendung von STL-Algorithmen (diese Algorithmen sind in der Regel gut optimiert. Eventuell können auch Kenntnisse über die verwendeten Container in die Implementierung einfließen).
    3. Die Funktionalität selbst schreiben (neben der wahrscheinlich schlechteren Effizienz, ist auch die Chance Fehler zu machen – man denke z.B. an invalidierte Iteratoren – wesentlich erhöht).
-

# Container speichern Kopien!

- Dass Container Kopien der Objekte speichern, hat unter Umständen unerwünschte Nebeneffekte:

1. Änderungen am Original werden im Container nicht reflektiert, bzw. vice versa

→ Beispiel

2. Bei Speicherung von Objekten einer erbenden Klasse kommt es zu slicing.

→ Beispiel

- Diese Effekte können vermieden werden, indem Pointer auf die Objekte gespeichert werden (Referenzen sind in C++ konstant und können daher nicht in Containern gespeichert werden). Das wiederum birgt die Gefahr, dass Pointer im Container auf Objekte verweisen, die schon zerstört wurden (dangling pointer). Daher sollte man hier auf smart pointer (`unique_ptr`, `shared_ptr`, `weak_ptr`) zurückgreifen.

## Effizienzvergleich `std::vector` vs. `Vector`

- Abgesehen davon, dass unsere Implementierung von `Vector` nicht den gesamten Umfang der Funktionalität von `std::vector` erreicht, verwenden `std::vector` und andere Container eine Reihe von Optimierungen:
- Freier Speicher wird nicht mit Default-Objekten gefüllt. Das `new` Statement zur Allokation von Speicher füllt das gesamte Array zunächst mit Default-Objekten. (Neben dem Performanceverlust bedeutet das auch, dass nur Objekte von Klassen mit Default-Konstruktor gespeichert werden können.)
- Kopien von Objekten werden so weit wie möglich vermieden. Inhalt von temporären Objekten kann direkt übernommen werden (move-Semantik), neue Objekte können direkt im passenden Speicherbereich anhand der Konstruktor-Parameter konstruiert werden (`emplace`) und nicht als Kopie eines anderen (eventuell temporären) Objekts, das als Parameter übergeben wird.
- Die Effizienz der STL Container lässt sich durch eigene Implementierungen kaum erreichen!



## 6. Exception safety

# Exception Safety

- Exception Safety umfasst Überlegungen zum Verhalten von Programmen im Fall, dass eine Exception geworfen wird.
- Klarerweise wird man sich wünschen, dass durch das Werfen einer Exception keine Inkonsistenzen in Objekten, Speicherlecks oder andere Fehlerzustände bewirkt werden.
- David Abrahams hat folgende Stufen von Garantien definiert, die die möglichen Fehler im Zusammenhang mit dem Werfen von Exceptions einschränken:
  - no throw guarantee
  - strong guarantee
  - basic guarantee
  - no guarantee

## No Throw Guarantee

- Ein Programmstück, das die no throw guarantee gibt, muss auf jeden Fall seine Aufgabe wie definiert erledigen. Eventuelle Exceptions müssen im Programmstück selbst gefangen und behandelt werden.
- Dies ist die stärkste Garantie, kann aber leider nicht immer gegeben werden (wie das Extrembeispiel eines Statements `throw runtime_error("Meldung");` zeigt).
- Wir haben schon in PR1 besprochen, dass Destruktoren keine Exception werfen sollten (um Probleme beim Stack-Unwinding zu vermeiden). Für Destruktoren sollte also immer die no throw guarantee gelten.
- Die no throw guarantee kann in C++ durch die `noexcept` Spezifikation bei der Funktionsdeklaration gegeben werden.



## Strong Guarantee

- Das Programmstück führt entweder seine Arbeit erfolgreich aus, oder hinterlässt – im Fall einer Exception – das Programm im ursprünglichen Zustand.
- Entspricht dem Transaktionsprinzip (alles oder nichts, z.B. Commit und Rollback)
- Kann oft nur mit großem Aufwand erreicht werden (z.B. Beim Einfügen eines Wertes in einen Container müsste der Ursprungszustand wieder hergestellt werden, was je nach genauer Ursache der Exception schwierig bis unmöglich sein könnte. Die Alternative, zunächst mit einer Kopie zu arbeiten und erst wenn alles funktioniert hat, die Kopie anstelle des Originals weiterzuverwenden, ist im Allgemeinen zu aufwendig).

## Basic Guarantee

- Das Programmstück hinterlässt die bearbeiteten Objekte in einem konsistenten Zustand und es gibt keine memory leaks.
- Diese Stufe kann in der Regel erreicht werden.
- Reicht leider oft nicht aus, da zwar die Objekte in einem konsistenten Zustand sind, aber es nicht klar ist, in welchem (z.B. wurde ein neues Objekt in den Container eingefügt oder nicht, ist der Container vielleicht jetzt eventuell leer, ...)

## No Guarantee

- Das Programmstück macht keinerlei Zusagen über den Zustand der bearbeiteten Objekte oder die Speicherfreigabe im Fall einer Exception.
- Wenig sinnvoll, sollte jedenfalls vermieden werden.

## Abhängigkeiten von Zusagen

- Ein Programmstück kann seine Zusagen nur unter der Bedingung einhalten, dass die verwendeten Funktionen ihre Zusagen ebenfalls erfüllen.
- Unter Umständen kann der Level der Zusagen von dem Level der Zusagen bereitgestellter Funktionen abhängen, z.B.: `vector<T>::erase` gibt die basic guarantee für beliebige Typen T (dabei wird von der STL vorausgesetzt, dass Destruktoren vom Typ T die no throw guarantee einhalten). Gilt für den Kopierkonstruktor und den Kopierzuweisungsoperator die no throw guarantee, kann diese auch für `erase` gegeben werden.
- Die genauen Garantien werden in der Dokumentation der jeweiligen Funktionen oder Methoden beschrieben.

## Beispiel: Vector<T>::reserve()

```
template<typename T>
void Vector<T>::reserve(typename Vector<T>::size_type new_sz) {
    if (new_sz <= max_sz) return;
    max_sz = new_sz;
    pointer new_buf = new value_type[max_sz]; //Eventuell std::bad_alloc Exception
    if (new_sz < sz) throw runtime_error("Buffer to small"); //Eventuell memory leak
    for (size_type i{0}; i < sz; ++i)
        new_buf[i] = values[i];
    delete[] values;
    values = new_buf;
}
```

- Diese Form der Funktion gibt keine Garantien. Es kann sowohl ein Memory-Leak entstehen, als auch ein inkonsistentes Objekt (`max_sz` stimmt nicht mehr mit der Größe der allozierten Memorybereichs überein)

## Beispiel: Vector<T>::reserve() bessere Version

```
template<typename T>
void Vector<T>::reserve(typename Vector<T>::size_type new_sz) {
    if (new_sz <= max_sz) return;
    if (new_sz < sz) throw runtime_error("Buffer to small");
    pointer new_buf = new value_type[new_sz]; //Eventuell std::bad_alloc Exception
    //Ab hier keine Exceptions mehr möglich, wenn Kopierzuweisungen und Destruktoren von T
    //die no throw guarantee einhalten
    for (size_type i{0}; i<sz; ++i)
        new_buf[i] = values[i];
    delete[] values;
    max_sz = new_sz;
    values = new_buf;
}
```

- Hier gilt die basic guarantee und sogar die strong guarantee. No throw guarantee kann nicht erreicht werden.



universität  
wien

## 7. Move Semantik

# Motivation

- Um die Effizienz von C++ Programmen zu erhöhen, versucht man, Kopien von Objekten wo immer möglich einzusparen.
- Copy elision bezeichnet Techniken, die der Compiler bei der Optimierung verwendet, um unnötige Kopier- (und Verschiebe-) operationen zu vermeiden.
- Historisch bedeutsam ist dabei die Optimierung der Rückgabe von Objekten durch eine Funktion: Return value optimization (RVO) und named return value optimization (NRVO)
- Die Optimierung geht so weit, dass der Compiler Konstruktoraufrufe (und die entsprechenden Destruktoraufrufe) auch dann vermeiden darf, wenn dadurch Seiteneffekte, die das beobachtbare Programmverhalten beeinflussen, verändert werden.
- Ab C++17 sind einige Optimierungen (z.B. RVO) vorgeschrieben.



## (N)RVO

```
T f() {  
    return T();  
}
```

```
T a{f()};
```

- Es wird kein temporäres Objekt erzeugt und dann nach `a` kopiert. Stattdessen wird das Objekt direkt in der Variablen `a` erzeugt.

```
T f() {  
    T t;  
    return t;  
}
```

```
T a{f()};
```

- `t` kann wie eine Referenz auf `a` behandelt werden (das ist nicht möglich, wenn `t` z.B. ein Parameter der Funktion ist).
-

## Idee hinter Move-Semantik

- An manchen Stellen wird ein Objekt nur für die Initialisierung eines anderen Objekts benötigt und danach nicht mehr verwendet. Solche Objekte müssen nicht aufwendig kopiert werden, man kann die interne Datenstruktur einfach übernehmen (“stehlen”).
- Es ist nur zu gewährleisten, dass auch nach dem “Stehlen” ein konsistentes Objekt (eventuell leer) zurückbleibt.
- Wir haben ein ähnliches Vorgehen schon beim Copy and Swap Idiom (oder Pattern) für den Kopierzuweisungsoperator kennengelernt:

## [PR1] Zuweisungsoperatoren (3) copy and swap

- Alternativ kann der Zuweisungsoperator auch so implementiert werden

```
String& String::operator= (String rightop) {  
    std::swap(maxlen, rightop.maxlen);  
    std::swap(len, rightop.len);  
    std::swap(buf, rightop.buf);  
    return *this;  
}
```

copy

swap  
(oft in einer eigenen  
swap Methode der  
Klasse realisiert)

- Eine konzise und simple Vorgehensweise, allerdings nicht effizient, wenn der schon vorhandene Speicher wiederverwendet werden könnte.

# Rvalue Reference

- Zur Unterstützung der Move-Operationen wurde ein neuer Datentyp(modifier) **rvalue reference** eingeführt.

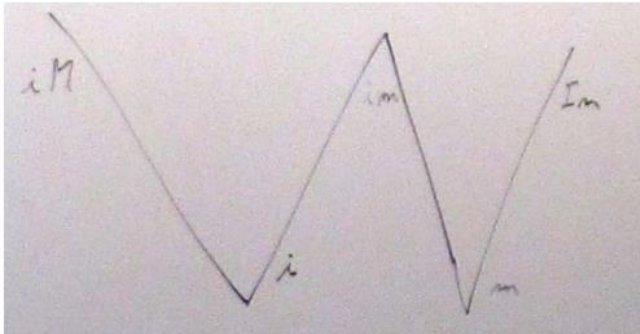
**T&& rvalRefT;**

- Als Funktionsparameter bindet dieser Datentyp nur mit Werten, die als Quelle für eine Move-Operation in Frage kommen (z.B. temporäre Variablen).

## lvalues und rvalues

- Die Begriffe l-value und r-value wurden ursprünglich in CPL verwendet, um Ausdrücke zu unterscheiden, denen etwas zugewiesen werden kann (left-hand-side) und solche, denen nichts zugewiesen werden kann (right-hand-side).
- lvalue wurde bei der Definition von C übernommen. Die Bedeutung hat sich aber von der Zuweisung entfernt (locator-value). rvalue hat sich dann in C++ für "nicht lvalue" etabliert.
- Mit der Einführung des `const` Schlüsselwortes in C++ wurde eine leichte Umdefinition notwendig (Konstante können nicht auf der linken Seite einer Zuweisung stehen, sind aber trotzdem lvalues).
- Mit C++11 wurde die Systematik erheblich erweitert und streng definiert, um die gewünschte Move-Semantik zu ermöglichen.

# Aktuelle Terminologie



Zwei interessante Eigenschaften eines Ausdrucks wurden identifiziert:

- hat Identität (i)
- move erlaubt (m)

Großbuchstaben stehen für: hat Eigenschaft nicht.



Um die alten Bedeutungen möglichst wenig zu verändern, wurden die Kategorien bezeichnet mit:

- lvalue
- glvalue (generalized lvalue)
- xvalue (eXpiring value)
- rvalue
- prvalue (pure rvalue)

# Aus drei mach fünf

- Die big three (bzw. rule of three) werden mit

Move-Konstruktor `T(T&&)` //kein const!

und Move-Kopierzuweisungsoperator `T& operator=(T&&)` //kein const!

zu den big five (bzw. rule of five).

If you write...

The compiler supplies...

	None	dtor	Copy-ctor	Copy-op=	Move-ctor	Move-op=
dtor	✓	•	✓	✓	✓	✓
Copy-ctor	✓	✓	•	✓	✗	✗
Copy-op=	✓	✓	✓	•	✗	✗
Move-ctor	✓	✗	Overload resolution will result in copying		•	✗
Move-op=	✓	✗			✗	•

Copy operations  
are independent...

Move operations  
are not.

# Move-Konstruktor typische Implementierung

```
Vector(Vector&& src) : Vector(min_alloc_size) {  
    swap(src.values, values);  
    swap(src.sz, sz);  
    swap(src.max_sz, max_sz);  
}
```

- Nicht ganz so typisch, da aufgrund von `min_alloc_size` in unserer Implementierung ein ganz leerer Vektor kein legales Objekt ist. Ohne diese Einschränkung ginge es effizienter:

```
Vector(Vector&& src) { //Einsparen von Allokation und überflüssigen Zuweisungen  
    values=src.values;  
    sz=src.sz;  
    max_sz=src.max_sz;  
    src.values=nullptr;  
    src.sz=src.max_sz=0;  
}
```



# Move-Kopierzuweisung typische Implementierung

```
Vector& operator=(Vector&& rhs) {  
    swap(rhs.values, values);  
    swap(rhs.sz, sz);  
    swap(rhs.max_sz, max_sz);  
    return *this;  
}
```

- Oder etwas effizienter:

```
Vector& operator=(Vector&& rhs) {  
    delete[] values;  
    values=rhs.values;  
    sz=rhs.sz;  
    max_sz=rhs.max_sz;  
    rhs.values=nullptr;  
    rhs.sz=rhs.max_sz=0;  
    return *this;  
}
```

## Achtung: T&& ergibt nicht unbedingt einen rvalue

```
void foo(X&& x)
{
    X anotherX = x; // calls X(X const & rhs)
}

X&& goo();
X x = goo(); // calls X(X&& rhs) because the thing on
              // the right hand side has no name
```

- Eine benannte Variable ist unabhängig vom Typ T&& ein lvalue. Damit wird verhindert, dass auf das nach dem Move undefinierte (aber konsistente) Objekt eventuell doch noch einmal zugegriffen wird.

```
X anotherX = x;
// x is still in scope!
```

## std::move

- `std::move` führt keine Move-Operation aus, sondern nur eine Typumwandlung von einem lvalue zu einem rvalue (auf den dann eine Move-Operation ausgeführt werden kann).

```
X anotherX = std::move(x);
```

```
Derived(Derived&& rhs) : Base(rhs) // wrong: rhs is an lvalue
{
    // Derived-specific stuff
}
```

```
Derived(Derived&& rhs) : Base(std::move(rhs)) // good, calls Base(Base&& rhs)
{
    // Derived-specific stuff
}
```

# Zu viel des Guten

- `std::move` ist nicht immer die beste Lösung:

```
X foo()  
{  
    X x;  
    // perhaps do something to x  
    return x;  
}
```

```
X foo()  
{  
    X x;  
    // perhaps do something to x  
    return std::move(x); // making it worse!  
}
```

- `std::move` verhindert in diesem Fall copy elision!

# Perfect forwarding

- Perfect forwarding treibt die Idee der Vermeidung von Kopien noch weiter.
- Statt des Anlegens eines temporären Objekts werden die Konstruktorparameter weitergereicht und das Objekt wird am Ort, wo es gebraucht wird, erstellt.
- Die diversen `emplace`-Methoden in den STL-Containern verwenden genau diese Strategie.
- Die Details der Implementierung von perfect forwarding sind komplex (universal reference, reference collapsing und special type deduction for rvalue references sind die Schlagworte, die die dazu verwendeten Sprachmittel beschreiben; eine detailliertere Beschreibung gibt <https://eli.thegreenplace.net/2014/perfect-forwarding-and-universal-references-in-c/#id1>).

## Beispiel

- Implementieren Sie eine einfache Kontoklasse (mit Kontostand und Rahmen, sowie Methoden zum Einzahlen, Abheben und Überweisen).
- Erweitern Sie die Klasse Person so, dass jede Person Verfüger über eine beliebige Anzahl von Konten sein kann. Jedes Konto hat andererseits maximal 4 Verfüger.
- Verhindern Sie, dass Objekte (z.B. durch Kopie in einen STL-Container) mehrfach auftreten. Verwenden Sie gegebenenfalls in den Containern nur Pointer auf die eigentlichen Objekte.
- Überladen Sie operator<< für die Klassen Person und Konto so, dass für eine Person die Konten, über die sie verfügt, ausgegeben werden und für ein Konto die jeweiligen Verfüger (Problem? Wie könnte das gelöst werden?).



universität  
wien

## 8. Smart Pointers

# Motivation

- Pointer sind ein mächtiges Werkzeug der Programmierung.
- Unter anderem sind sie notwendig, um mit dynamischem Speicher zu arbeiten.
- David Wheeler (Mit)Erfinder der Subroutine:
- *“All problems in computer science can be solved by another level of indirection...”*



# Probleme mit Pointern

- “...*But that usually will create another problem.*”
- Oft auch “...*except for the problem of too many layers of indirection.*”
- *Pointers are like jumps, leading wildly from one part of the data structure to another. Their introduction into high-level languages has been a step backward from which we may never recover* (C.A.R. Hoare)
- Nicht oder falsch initialisierte Pointer
- Dangling Pointer
- C++ spezifisch:
  - Wer besitzt das Objekt, auf das gezeigt wird (und ist damit für dessen Zerstörung zuständig)?
  - Wird auf ein einzelnes Objekt oder auf ein Array gezeigt?
  - Was ist die richtige Art, das Objekt zu zerstören (`delete`, `delete[]` oder vielleicht eine spezielle Methode)?

# Smart Pointer

- Wir haben bereits RAII kennen gelernt, um die Probleme mit der Ressourcenverwaltung (Allokation und Freigabe von Ressourcen) zu lösen.
- Smart Pointer sind eine vorprogrammierte RAII-Lösung.
- ~~`std::auto_ptr`~~ (deprecated)
- `std::unique_ptr` (exklusives Eigentum)
- `std::shared_ptr` (geteiltes Eigentum, reference count)
- `std::weak_ptr` (kein Eigentum)
- Alle diese Typen sind in der Library memory (`#include<memory>`) definiert.

## `std::unique_ptr`

- Besitzt das Objekt, auf das gezeigt wird, exklusiv.
- Fast kein Overhead im Vergleich zu einfachen Pointern (raw pointer, naked pointer). Es wird nur ein Pointer auf das Objekt gespeichert und die Dereferenzierung (`operator*`) ist inline und erzeugt somit den gleichen Code wie die Dereferenzierung eines einfachen Pointers.
- Kopien sind verboten (da es sonst mehrere Besitzer geben würde)
- Move überträgt den Besitz auf das Ziel der Move-Operation
- Das Objekt wird im Destruktor von `std::unique_ptr` zerstört.
- Der Pointer “weiß”, ob ein einzelnes Objekt oder ein Array referenziert wird, da es neben `std::unique_ptr<T>` auch eine partielle Spezialisierung `std::unique_ptr<T[]>` gibt.
- Zerstört wird defaultmäßig durch `delete` bzw. `delete[]`.
- Es können auch Funktionen angegeben werden, die die Zerstörung des Objekts übernehmen (custom deleter), dadurch wird aber der Overhead im Vergleich zu einfachen Pointern (raw pointer, naked pointer) größer

## std::unique\_ptr Beispiel (1)

```
unique_ptr<int> ip{new int}; //wird automatisch mit delete freigegeben
unique_ptr<int[]> ia{new int[4]{1,2,3,4}}; //Wird automatisch mit delete[] freigegeben
                                     //Verwendung eines STL-Containers waere besser
unique_ptr<vector<int>> ivp{new vector<int>{1,2,3,4}};
unique_ptr<Angestellte> ap{new Angestellte("Maria", "Mayr", 21, 1010, 299, 100000)};

*ip = 10;
cout << *ip + 5 << '\n';    //Ausgabe 15
//cout << ip[3] << '\n';    //nicht erlaubt

//cout << *ia;                //nicht erlaubt
cout << ia[2] << '\n';        //Ausgabe 3

cout << (*ivp)[3] << '\n';    //Ausgabe 4
cout << ivp->at(1) << '\n';    //Ausgabe 2

cout<< ap->get_name() << '\n'; //Ausgabe Mayr Maria

unique_ptr<Angestellte> ap1{ap}; //ap1 ist nun Eigentuermer  ap ist nullptr
unique_ptr<Angestellte> ap2;
ap2 = ap; //beide sind nun nullptr
//*ap; //undefiniertes Verhalten
```

**Compilerfehler**

## std::unique\_ptr Beispiel (2)

```
unique_ptr<int> ip{new int}; //wird automatisch mit delete freigegeben
unique_ptr<int[]> ia{new int[4]{1,2,3,4}}; //Wird automatisch mit delete[] freigegeben
//Verwendung eines STL-Containers waere besser
unique_ptr<vector<int>> ivp{new vector<int>{1,2,3,4}};
unique_ptr<Angestellte> ap{new Angestellte("Maria", "Mayr", 21, 1010, 299, 100000)};

*ip = 10;
cout << *ip + 5 << '\n'; //Ausgabe 15
//cout << ip[3] << '\n'; //nicht erlaubt

//cout << *ia; //nicht erlaubt
cout << ia[2] << '\n'; //Ausgabe 3

cout << (*ivp)[3] << '\n'; //Ausgabe 4
cout << ivp->at(1) << '\n'; //Ausgabe 2

cout<< ap->get_name() << '\n'; //Ausgabe Mayr Maria

unique_ptr<Angestellte> ap1{std::move(ap)}; //ap1 ist nun Eigentuermer ap ist nullptr
unique_ptr<Angestellte> ap2;
ap2 = std::move(ap); //beide sind nun nullptr
//*ap; //undefiniertes Verhalten
```

## `std::unique_ptr` weitere Methoden

- `pointer release() noexcept` Besitz aufgeben
- `void reset(pointer) noexcept` Ein anderes Objekt in Besitz nehmen
- `pointer get() const noexcept` liefert den Pointer auf das besessene Objekt
- `explicit operator bool() const noexcept` `false`, wenn kein Objekt besessen wird (`nullptr`)

## `std::shared_ptr`

- Teilt das Objekt, auf das gezeigt wird, mit anderen Besitzern.
- Overhead zu einfachen Pointern durch Referenzzähler (Speicher) und dessen Inkrementierung/Dekrementierung (Laufzeit)
- Kopien sind erlaubt (Inkrementierung des Referenzzählers)
- Move überträgt den Besitz auf das Ziel der Move-Operation (Referenzzähler wird nicht verändert)
- Das Objekt wird zerstört, sobald der letzte Besitzer den Besitz aufgibt
- Es können nur Pointer auf einzelne Objekte gespeichert werden, nicht auf Arrays (ab C++17 sind auch Arrays möglich)
- Zerstört wird defaultmäßig durch `delete` (bzw. ab C++17 eventuell durch `delete[]`). Custom deleters sind möglich.

## `std::shared_ptr` weitere Methoden

- `void reset(T*) noexcept` Ein anderes Objekt in Besitz nehmen
- `T* pointer get() const noexcept` liefert den Pointer auf das besessene Objekt
- `long use_count() const noexcept` liefert die Anzahl der Besitzer des besessenen Objekts
- `explicit operator bool() const noexcept` false, wenn kein Objekt besessen wird (nullptr)



## `std::weak_ptr`

- Ähnlich wie `std::shared_ptr`, allerdings kann ein `std::weak_ptr` auch dangling sein (das Objekt auf das gezeigt wird, ist schon freigegeben).
- Zum Zugriff muss mittels `lock()` ein `std::shared_ptr` erzeugt werden.

## `std::weak_ptr` weitere Methoden

- `void reset(T*) noexcept` Ein anderes Objekt in Besitz nehmen
- `long use_count() const noexcept` liefert die Anzahl der Besitzer des besessenen Objekts
- `bool expired() const noexcept` true, wenn das Objekt schon gelöscht wurde
- `std::shared_ptr<T> lock() const noexcept` liefert einen passenden shared Pointer (Default Pointer ohne Objekt, wenn `expired()` true ist)

## `std::make_unique` und `std::make_shared`

- Bieten eine etwas vereinfachte Syntax an, um smart pointer zu erzeugen  
z.B.: `auto p = make_unique<Angestellte>();`  
(eventuell angegebene Parameter werden mittels perfect forwarding an den Konstruktor weitergereicht).
- Verhindern dadurch Fehler, die bei Übernahme von raw pointern entstehen können:
  - Mehrmals denselben raw pointer zu verwenden, um einen `unique_` oder `shared_ptr` zu erstellen.
  - Exception unsafe code in speziellen Sonderfällen zu erzeugen z.B.:  
`f(std::shared_ptr<T>{new(T)}, g()); //potentielles memory leak`
- Sind daher in der Regel vorzuziehen.

## `std::make_unique`, `std::make_shared` Kontraindikationen

- Können nicht verwendet werden, wenn custom deleter verwendet werden sollen
- Können nicht zur Initialisierung mit braced initializer list verwendet werden
- `std::make_unique` ist erst ab C++14 verfügbar
- Für `std::shared_ptr` gilt außerdem (da der Kontrollblock bei `std::make_shared` aus Effizienzgründen gemeinsam mit dem referenzierten Objekt alloziert wird):
  - vertragen sich nicht mit Klassen, die ihre eigenen `new` und `delete` Methoden definieren
  - verhindern die Freigabe von Objekten, deren Referenzzähler schon auf 0 gegangen ist.

## Rule of zero

- Nachdem wir uns über die rule of three zur rule of five vorgearbeitet haben, betrachten wir nun die neueste Denkschule: rule of zero (Peter Somerlad).
- *Write your classes in a way that you do not need to declare/define neither a destructor, nor a copy/move constructor or copy/move assignment operator*
- *Use smart pointers & standard library classes for managing resources*

*Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership*

## Rule of zero (virtual destructor) 1

- Klassen, deren Objekte über einen Pointer der Basisklasse freigegeben werden (Faustregel: Alle Klassen, die zumindest eine virtuelle Methode anbieten) brauchen einen virtuellen Destruktor.

```
class C {  
    virtual void foo();  
public:  
    virtual ~C() = default;  
}
```

- In C++11 werden aber dadurch die Move-Operationen deaktiviert.

## Rule of zero (virtual destructor) 2

```
class C {  
    virtual void foo();  
public:  
    virtual ~C() = default;  
    C(C&&) = default;  
    C& operator= (C&&) = default;  
}
```

- C++14 hat die Regeln erneut geändert und es würden nun Copy-Konstruktor und Copy-Zuweisungsoperator in diesem Fall nicht mehr erzeugt.

## Rule of zero (virtual destructor) 3

```
class C {  
    virtual void foo();  
public:  
    virtual ~C() = default;  
    C(const C&) = default;  
    C& operator= (const C&) = default;  
    C(C&&) = default;  
    C& operator= (C&&) = default;  
}
```

- ... und wir wären zurück bei 5. (Eventuell wird noch ein Defaultkonstruktor benötigt.)
- Das kann vermieden werden:



## Rule of zero (virtual destructor) 4

- *Use smart pointers & standard library classes for managing resources*

Durch Verwenden von `shared_ptr` statt raw pointer wird der korrekte Destruktor auch aufgerufen, wenn der Destruktor nicht virtuell ist!  
(Warum nicht auch bei `unique_ptr`?)

## std::enable\_shared\_from\_this

- Will man einen `shared_ptr` auf das `this` Objekt erhalten, dann sollte man diesen nicht direkt erstellen. (Falls es bereits einen anderen `shared_ptr` auf das Objekt gibt, würde das zu undefiniertem Verhalten führen.)
- Falls es schon einen `shared_ptr` auf das Objekt gibt, kann diese Methode verwendet werden:

```
class C: public std::enable_shared_from_this<C>
{
...
}
```

- In der Klasse kann nun die Methode `shared_from_this()` verwendet werden, um einen zusätzlichen `shared_ptr` auf das Objekt zu erhalten. Wird das Objekt noch nicht durch einen `shared_ptr` verwaltet, so führt dies zu undefiniertem Verhalten.



universität  
wien

## 9. Vererbung

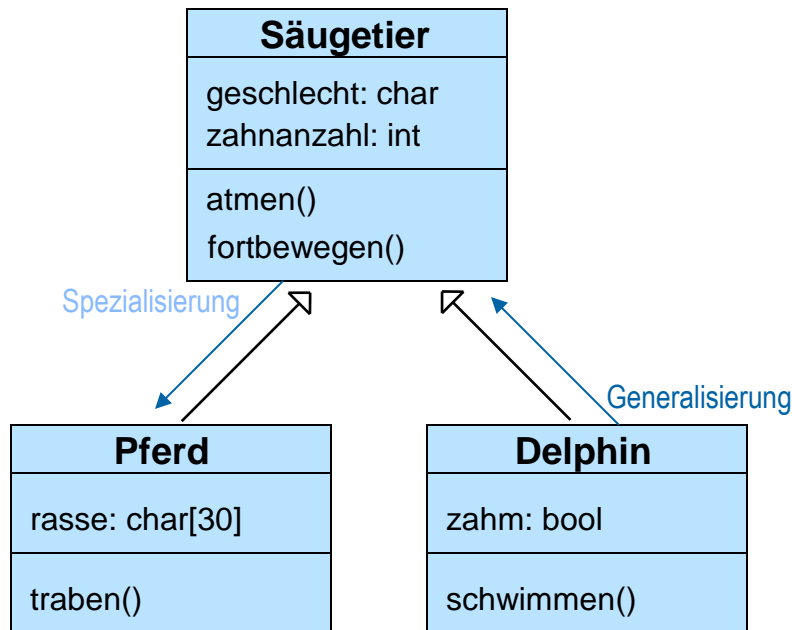
Inheritance

# Vererbung

- Vererbung bildet mit Datenkapselung und Polymorphismus die drei grundlegenden Konzepte der Objektorientierung
- Sie erlaubt die einfache Wiederverwendung von bereits produziertem Code (code reuse) und damit
  - **Einfachere Programmierung**
  - **Einfacheres Fehlersuchen und Testen**
- Üblicherweise erbt eine spezialisierte Klasse (Unterklasse) von einer generelleren Klasse (Oberklasse, Basisklasse). Dabei gilt eine „is-a“ Beziehung (jedes Element der Unterklasse ist auch ein Element der Oberklasse)

# Beispiel

## •UML



## C++

```

class Saeugetier {
    char geschlecht;
    int zahnanzahl;
public:
    void atmen();
    void fortbewegen();
};

class Pferd: public Saeugetier {
    char rasse[30];
public:
    void traben();
};

class Delphin: public Saeugetier {
    bool zahm;
public:
    void schwimmen();
};
  
```

Ein Pferd ist eine spezielle Art Säugetier und hat damit alle Eigenschaften und Methoden der Klasse Säugetier, sowie zusätzlich diejenigen der Klasse Pferd (analog für Delphin)

# Substitutionsprinzip (Liskov)

- Ein Objekt der abgeleiteten (erbenden Klasse) kann an jeder beliebigen Stelle für ein Objekt der Basisklasse eingesetzt werden.

z.B.:

```
void f(Saeugetier);
```

```
Saeugetier s;  
Pferd p;  
s.atmen(); //OK  
p.traben(); //OK  
f(p); //OK  
s=p; //OK  
s.atmen(); //OK  
s.traben(); //nicht erlaubt  
p=s; //nicht erlaubt
```

## Vererbung ohne Substitutionsprinzip

- Die Mechanismen, die von Programmiersprachen zur Vererbung zur Verfügung gestellt werden, erlauben meist auch ein „Unterlaufen“ des Substitutionsprinzips.
- Das kann manchmal nutzbringend verwendet werden, führt aber bei Unvorsichtigkeit zu Problemen. Sollte daher nur eingesetzt werden, wenn wirklich notwendig.

z.B. Einschränkung:

```
class Untier: private Saeugetier {  
};
```

```
Untier u;  
u.atmen();           //nicht erlaubt (private Methode)  
Saeugetier t=u;      //nun auch nicht mehr erlaubt
```

Private Vererbung: Alle Methoden und Instanzvariablen der Oberklasse sind in der Unterklasse private  
Default Vererbung ist private für Klassen und public für Structs

# Überladen von Methoden der Basisklasse

- Oft ist es notwendig, in der erbenden Klasse das Verhalten der Basisklasse nicht komplett zu übernehmen, sondern zu adaptieren.

z.B.: Delphine müssen zum Atmen erst auftauchen.

```
class Delphin: public Saeugetier {  
    bool zahm;  
public:  
    void schwimmen();  
    void auftauchen();  
    void atmen() {auftauchen(); ...}  
};
```

```
Saeugetier s;  
Delphin d;  
d.atmen(); //Delphin::atmen  
s=d;  
s.atmen(); //Saeugetier::atmen! Information, dass es sich um  
//einen Delphin handelt ist verloren gegangen
```

Override: Delphin::atmen überlagert alle (eventuell überladenen) Methoden atmen der Basisklasse (**Saeugetier::atmen**)

Dies kann bei Bedarf durch eine using-Deklaration rückgängig gemacht werden:

**using Saeugetier::atmen;**



## Aufruf der Methode der Basisklasse

- Wie in unserem Beispiel ist es sehr oft nützlich, die Funktionalität der Basisklasse in einer überlagerten Methode weiterhin zu nutzen.

```
class Delphin: public Saeugetier {  
    bool zahm;  
public:  
    void schwimmen();  
    void auftauchen();  
    void atmen() {auftauchen(); Saeugetier::atmen();}  
};
```

Taucht zunächst auf und  
führt dann „normale“ Atmung  
aus

## Implizite Pointerkonversion

- Normalerweise findet zwischen Zeigern unterschiedlichen Typs keine implizite Typumwandlung statt:

```
int *x; double *y = x;
```

- Ein Zeiger (eine Referenz) auf eine abgeleitete Klasse kann aber implizit auf einen Zeiger (eine Referenz) auf eine Basisklasse umgewandelt werden:

```
Delphin d; Saeugetier *sp = &d; Saeugetier &sr = d;
```

- Diese Art der Umwandlung wird (als einzige) auch bei catch-Statements implizit durchgeführt. Man kann also z. B. mit `catch (Saeugetier&)` alle Exceptions vom Typ Saeugetier und allen davon erbenden Klassen fangen.

## dynamic\_cast

- Die umgekehrte Richtung der Typumwandlung kann explizit erzwungen werden, ist aber eventuell gefährlich:

```
(static_cast<Delphin *>(sp)) ->schwimmen();  
(static_cast<Pferd &>(sr)).traben();
```

Syntaktisch korrekt, aber Objekt ist ein Delphin und kann nicht traben. Undefinedes Verhalten zur Laufzeit.

- dynamic\_cast** führt eine Laufzeitüberprüfung durch, ob das Objekt auch wirklich den erforderlichen Typ hat und liefert im Fehlerfall entweder **nullptr** (für Pointer) oder wirft eine Exception vom Typ **std::bad\_cast** (für Referenzen).

```
(dynamic_cast<Pferd &>(sr)).traben(); // Exception
```

# Polymorphismus

- Die Auswirkung eines Funktionsaufrufs hängt von den Typen der beteiligten Objekte ab.
- Eine bereits bekannte Möglichkeit, dies zu erreichen ist Überladen (ad-hoc Polymorphismus):  
`f(Saeugetier), f(Pferd), f(Delphin)`
- Vererbung bietet eine weitere Möglichkeit:

```
class Saeugetier {  
    ...  
    virtual void fortbewegen();  
};  
  
class Pferd: public Saeugetier {  
    ...  
    void traben();  
    virtual void fortbewegen() {traben();}  
};  
  
class Delphin: public Saeugetier {  
    ...  
    void schwimmen();  
    virtual void fortbewegen() {schwimmen();}  
};
```

Muss nicht  
mehr explizit  
angegeben  
werden

```
Pferd *p = new Pferd;  
Saeugetier *sp = p; //erlaubt  
sp->fortbewegen(); //trabt
```

```
Delphin *d = new Delphin;  
Saeugetier &sr = *d;  
sr.fortbewegen(); //schwimmt
```

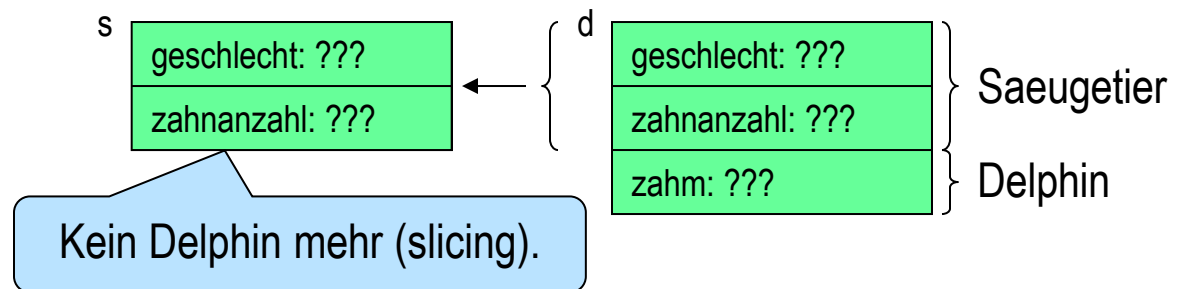
```
Saeugetier s=*p;  
s.fortbewegen(); //Saeugetier::fortbewegen()
```

Polymorphes  
Verhalten der  
Objekte in  
C++ nur bei  
Zeigern und  
Referenzen.

## Warum nur mit Pointern oder Referenzen? (1)

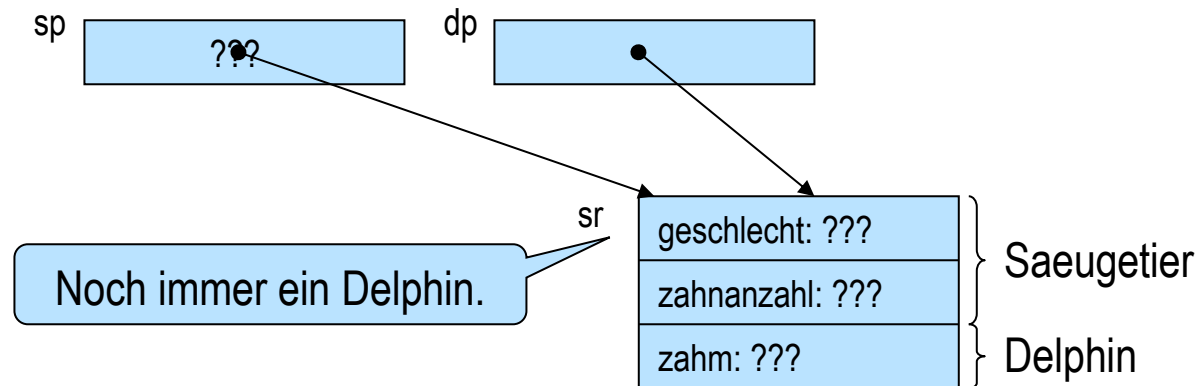
- Eine Variable in C++ entspricht einer Adresse, an der das Objekt im Speicher zu finden ist.
- Zuweisung entspricht einer Kopie von einer Adresse zu einer anderen.
- Objekte sind „modular“ aufgebaut (Eigene Instanzvariablen, Instanzvariablen der direkten Basisklasse und so weiter).
- Bei einer Zuweisung eines Objekts an eine Variable vom Typ einer Basisklasse, werden nur die relevanten Objektteile kopiert.

```
Saeugetier s;  
Delphin d;  
s = d;
```



## Warum nur mit Pointern oder Referenzen? (2)

```
Saeugetier *sp;  
Delphin *dp = new Delphin;  
sp = dp;  
Saeugetier &sr = *dp;
```



## override Spezifikation (Seit C++11)

- Damit ein Override stattfindet, müssen die Funktionssignaturen genau übereinstimmen. Um Irrtümer zu vermeiden kann explizit **override** angeführt werden:

```
class A {  
    virtual int f() const;  
};  
class B : public A {  
    int f(); //kein override (auch ein weiteres virtual hilft nicht)  
    int f(double); //kein override  
    int f() const; //override  
}  
class C : public A {  
    int f() override; //Fehler  
    int f(double) override; //Fehler  
    int f() const override; //OK  
}
```

# Abstrakte Klasse

- Oft ist es nicht möglich, das Verhalten einer virtuellen Methode in der Basisklasse zu definieren. So haben z.B. Säugetiere keine gemeinsame Art sich fortzubewegen.
- Man kann die Methode dann als „pure virtual“ (abstrakt) kennzeichnen. Das heißt, sie wird erst in den abgeleiteten Klassen implementiert.
- Eine Klasse mit (mindestens einer) pure virtual Methode kann keine Objektinstanzen haben (z.B. es gibt kein Säugetier an sich, sondern es muss immer eine bestimmte Unterart sein). Eine solche Klasse wird als „abstrakt“ (abstract) bezeichnet.

```
class Saeugetier {  
    ...  
    virtual void fortbewegen() =0;  
};
```

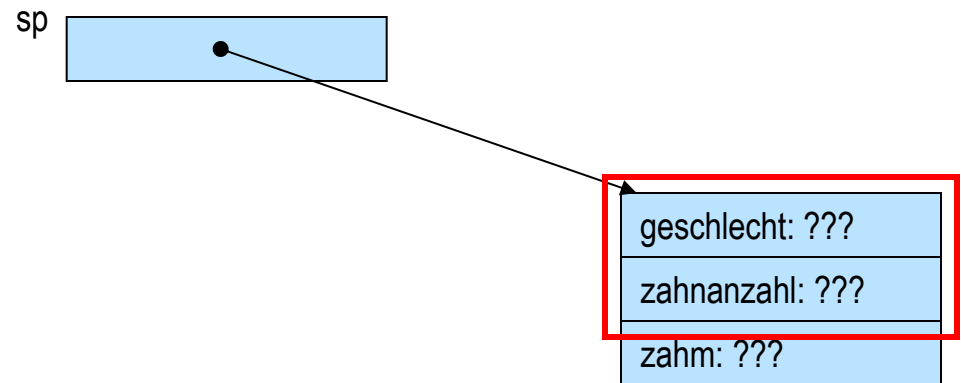
Methode wird in dieser Klasse nicht implementiert. Erbende Klassen müssen (so sie nicht auch abstrakt sind) die Methode implementieren



## Virtueller Destruktor (1)

- Wird ein dynamisch erzeugtes Objekt über eine Pointer auf eine Basisklasse gelöscht, „weiß“ der Compiler die Größe des zu löschenden Objekts nicht.

```
Saeugetier *sp = new Delphin;  
delete sp;
```



## Virtueller Destruktor (2)

- Lösung: virtueller Destruktor

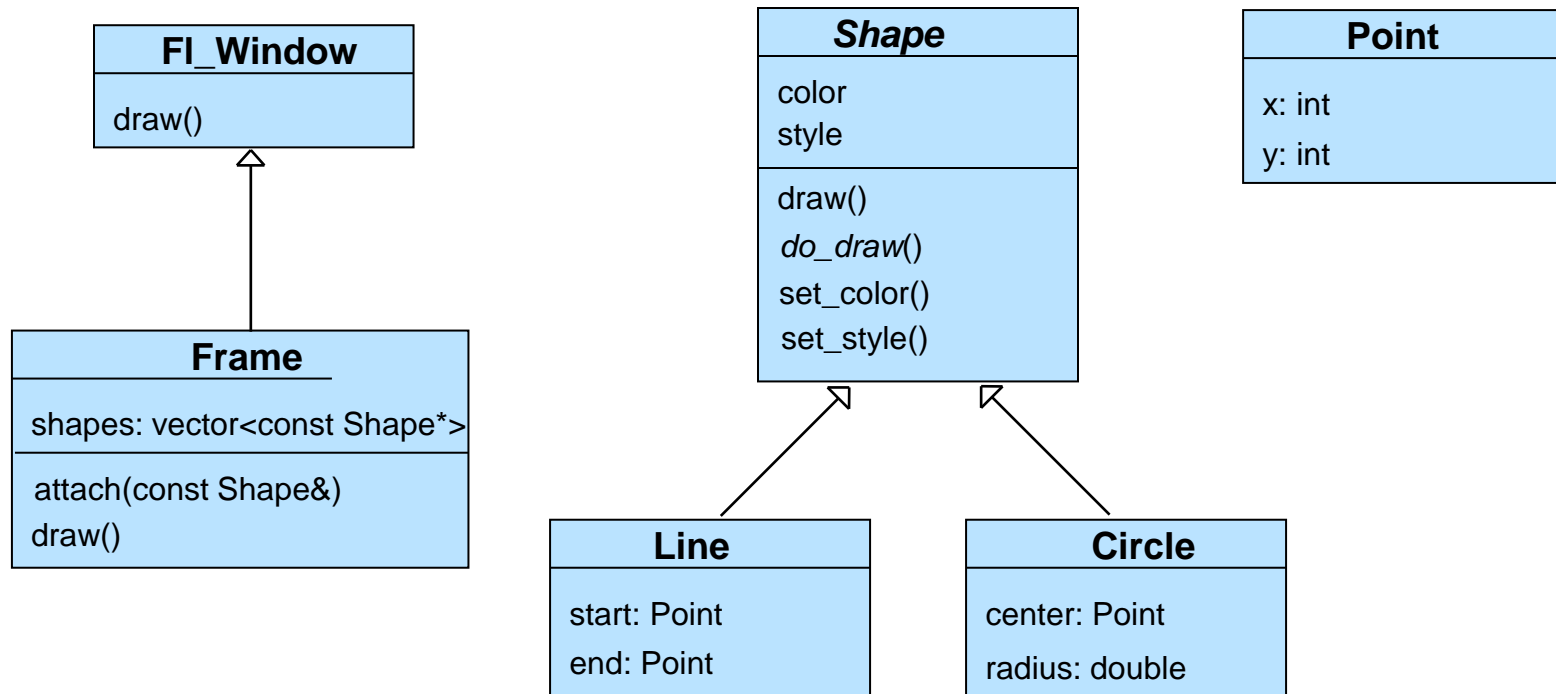
```
class Saeugetier {  
    ...  
    virtual ~Saeugetier();  
};
```

Damit haben alle erbbenden Klassen ebenfalls einen virtuellen Destruktor (ob dort nochmals definiert oder nicht) und es wird bei polymorpher Verwendung immer der richtige Destruktor aufgerufen.

- Faustregel: In abstrakten Klassen immer einen virtuellen Destruktor definieren (meist schon in Klassen, sobald sie eine virtuelle Funktion haben).

## Beispiel: Einfache Grafikapplikation

- Kann unterschiedliche Formen (Shapes) zeichnen.



# Struct Point

```
struct Point {  
    int x;  
    int y;  
    Point(int x, int y) : x(x), y(y) {};  
};
```

## Klasse Frame

- Erbt von Fl\_Window. Für dieses wird draw aufgerufen, wann immer der Inhalt des Fensters neu gezeichnet werden muss. Frame macht zuerst alles, was Fl\_Window macht und ruft dann draw für jede assoziierte Form auf.

```
class Frame : public Fl_Window {
    std::vector<const Shape*> shapes;
public:
    Frame(int x, int y, const char* text) : Fl_Window(x, y, text) {}
    void draw() override {
        Fl_Window::draw();
        for (const auto& shape : shapes) shape->draw();
    }

    void attach(const Shape& s) {
        shapes.push_back(&s);
    }
};
```

# Klasse Shape

- Ruft, wenn draw() aufgerufen wird, die Methode do\_draw() in der passenden erbbenden Klasse auf. Erlaubt Setzen von Farbe und Linien-Stil.

```
class Shape {
    Fl_Color color {FL_BLACK};
    int style {0};
public:
    virtual ~Shape() {}
    void draw() const {
        Fl_Color old_color {fl_color()};
        fl_color(color);
        fl_line_style(style); //style cannot be queried - will be reset to default
        do_draw();
        fl_line_style(0); //default
        fl_color(old_color);
    }
    virtual void do_draw() const = 0;
    void set_color(Fl_Color color) {
        this->color = color;
    }
    void set_style(int style) {
        this->style = style;
    }
};
```

# Klassen Line und Circle

- "Zeichnen sich selbst".

```
class Line : public Shape {
    Point start;
    Point end;
public:
    Line(Point start, Point end) : start {start}, end {end} {}
    void do_draw() const override {
        fl_line(start.x, start.y, end.x, end.y);
    }
};

class Circle : public Shape {
    Point center;
    double radius;
public:
    Circle(Point center, double radius) : center {center}, radius {radius} {}
    void do_draw() const override {
        fl_circle(center.x, center.y, radius);
    }
};
```

# main()

```
int main() {  
    Frame frame {450,220,"Graph"};  
    Line line {Point{0, 0}, Point{100, 100}};  
    frame.attach(line);  
    Circle c1 {Point{0, 0}, 25};  
    frame.attach(c1);  
    Circle c2 {Point{100, 100}, 25};  
    frame.attach(c2);  
    c1.set_color(FL_RED);  
    frame.show();  
    return Fl::run();  
}
```





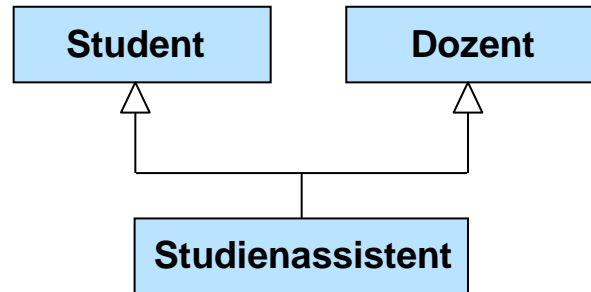


# 10. Mehrfachvererbung

- Multiple Inheritance

## Multiple Inheritance

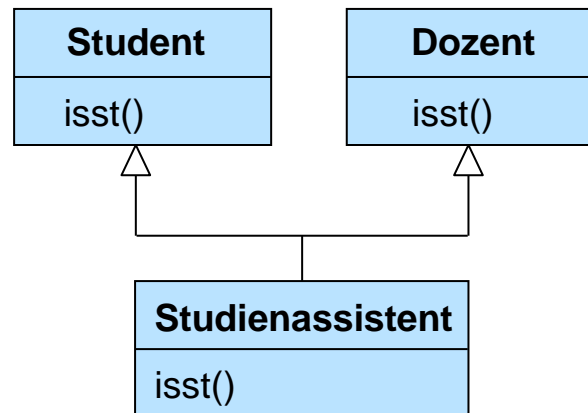
- Manchmal ist es vorteilhaft oder notwendig, dass eine Klasse die Eigenschaften mehrerer Basisklassen übernimmt. In Sprachen, die multiple inheritance unterstützen, lässt sich das direkt ausdrücken.



```
class Studienassistent : public Student, public Dozent {  
    ...  
};
```

## Multiple Inheritance Probleme (1)

- Wird dieselbe Methode von mehreren Klassen geerbt, so muss der Aufruf entsprechend qualifiziert werden, um Mehrdeutigkeiten zu vermeiden:



- In UML ist die Semantik nicht genau definiert. In C++ gilt:
- Um die ererbten Funktionen aufzurufen, muss `student::isst()` bzw. `dozent::isst()` verwendet werden. (Analoges gilt für Instanzvariablen.)
- Der override betrifft die `isst`-Methoden aus beiden Basisklassen!

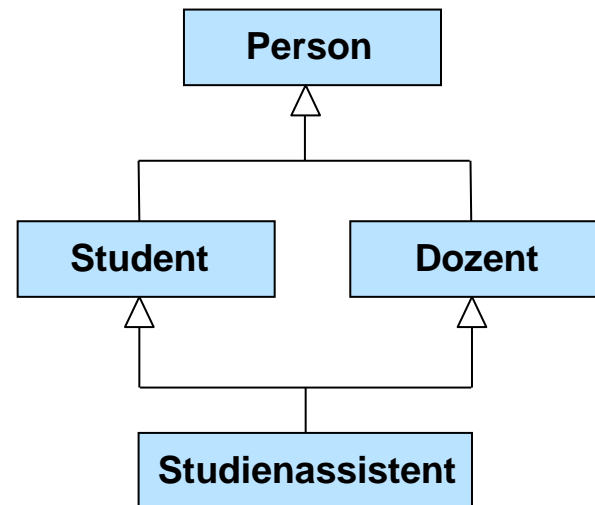
## Multiple Inheritance Probleme (2)

- Will man beide Methoden der Basisklassen unabhängig voneinander override, so kann man entsprechende Klassen dazwischen schieben:

```
class Student {  
    virtual void isst();  
    ...  
};  
class Dozent {  
    virtual void isst();  
    ...  
};  
class Student_help : public Student {  
    virtual void student_isst() = 0;  
    void isst() {student_isst();}  
};  
class Dozent_help : public Dozent {  
    virtual void dozent_isst() = 0;  
    void isst() {dozent_isst();}  
};  
class Studienassistent : public Student_help, public Dozent_help{  
    void student_isst() {...}  
    void dozent_isst() {...}  
};
```

## Multiple Inheritance Probleme (3)

- Diamond of Death:



- In C++ enthält ein Studienassistent nun zwei Personen-Basisobjekte.

# Virtual Inheritance

- Das Vervielfachen der Basisobjekte kann durch virtual inheritance vermieden werden:

```
class Person {  
    ...  
};  
class Student : virtual public Person {  
    ...  
};  
class Dozent : virtual public Person {  
    ...  
};  
class Studienassistent : public Student, public Dozent {  
    ...  
};
```

- Virtual inheritance ist weniger effizient (sowohl bezüglich Laufzeit als auch bezüglich Speicherbedarf) und hat auch Besonderheiten beim Initialisieren und Zuweisen von Objekten.
- (Diese Probleme kommen nicht zum Tragen, wenn die Basisklasse keine Daten und nur (pure) virtuelle Funktionen enthält (ABC abstract base class)).

## Virtual Inheritance Besonderheiten

- Virtuelle Basisklassen werden vor allen anderen Basisklassen initialisiert (Konstruktoraufruf).
- Virtuelle Basisklassen müssen in der most derived class des Objekts (sozusagen die Klasse mit der das Objekt erzeugt wurde) initialisiert werden. Fügt man eine neue Klasse zur Hierarchie hinzu, muss man auch an die Initialisierung der virtuellen Basisklassen denken.
- Die automatisch generierten Zuweisungsoperatoren (copy und move), weisen eventuell mehrfach zu (der Standard lässt das undefiniert). Besonders beim Move-Zuweisungsoperator ist das problematisch, da jede Zuweisung ja das Source-Objekt "zerstört". Die zweite Zuweisung würde dann die sinnlosen Daten aus dem Source-Objekt übernehmen.

## Alternativen zu Virtual Inheritance

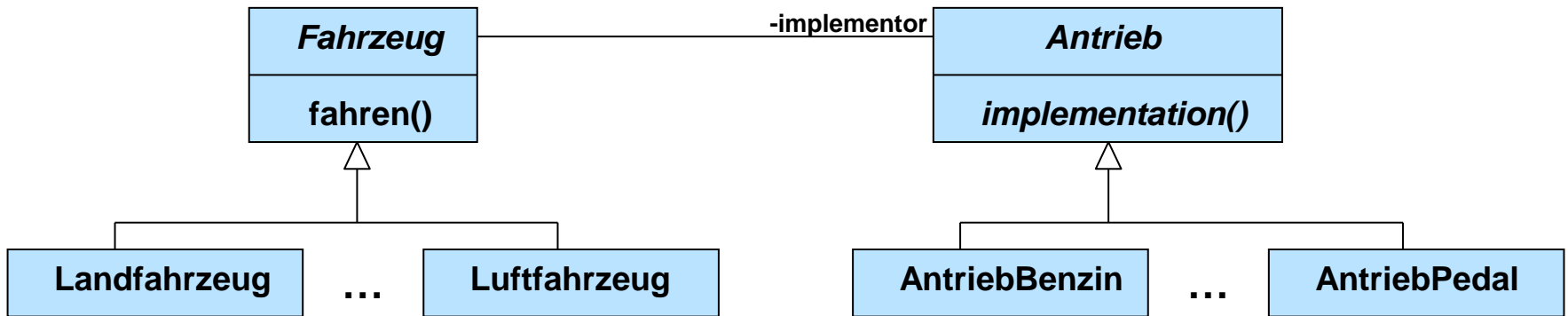
- Statt virtual inheritance können auch (zumindest) verschachtelte Generalisierung (nested generalization) und das Bridge Pattern verwendet werden.
- Als Beispiel seien zwei Vererbungshierarchien für Fahrzeuge zu kombinieren.
  1. Medium: Land-, Wasser-, Luftfahrzeug
  2. Antrieb: Benzin-, Strom-, Wind-, Pedal-getrieben
- Selbstverständlich bietet sich multiple inheritance als eine Lösungsmöglichkeit sofort an.



# Nested Generalization

- Eine der beiden Hierarchien wird als übergeordnete Hierarchie identifiziert und man implementiert die entsprechenden Kombinationen mit single inheritance (z.B. Medium sei die dominierende Hierarchie)
  - Land – LandBenzin, LandStrom, ...
  - Wasser – WasserBenzin, WasserStrom, ...
  - ...
  - Die Anzahl der (möglichen Unterklassen) ist das Produkt der Anzahlen der Klassen in beiden Hierarchien.
  - Dafür kann (wie auch bei multiple inheritance) sehr fein gewählt werden, welche Kombinationen sinnvoll sind (man muss z.B. keine Klasse für ein pedalgetriebenes Raumschiff anbieten).
  - Ist, im Gegensatz zur multiple inheritance, asymmetrisch
-

# Bridge Pattern



- Die Anzahl der zu implementierenden Unterklassen) ist dieSumme der Anzahlen der Klassen in beiden Hierarchien.
- Diese Konstruktion ist extrem flexibel, da die Verbindungen zur Laufzeit beliebig gesetzt (und sogar geändert) werden können.
- Nicht sinnvolle Kombinationen sind aber nur zur Laufzeit (und auch da nur schwer) zu verhindern.
- Wie nested generalization asymmetrisch



universität  
wien

# 11. Klassendesign und Vererbung

## Vererben oder ...?

- Allgemein wird Vererbung gerne überstrapaziert.
- Um möglichst alle Probleme auszuschließen sollte Vererbung nur für is-a Beziehungen (LSP ist erfüllt) verwendet werden.
- Für is-implemented-in-terms-of kann auch Vererbung verwendet werden, diese sollte aber private sein.
- Als Alternative steht regelmäßig die Komposition zur Verfügung. Dabei wird ein Objekt der Klasse, das die eigentliche Arbeit macht, als Komponente (Instanzvariable, meist auch nur ein Pointer auf das enthaltene Objekt) gespeichert. Funktionsaufrufe werden an dieses Objekt delegiert.
- Im Allgemeinen gilt: Composition over Inheritance

# Vererbung vs. Komposition

Vererbung	Komposition
starr (zur Übersetzungs-Zeit)	flexibel (zur Laufzeit)
nur ein Basisobjekt (abgesehen von MI)	beliebig viele Objekte, an die delegiert werden kann
effizient	Delegation notwendig
gesamtes Interface wird übernommen	beliebige Teile des Interfaces können „übernommen“ werden
polymorphes Verhalten via Pointer/Referenz auf die Basisklasse	Polymorph, wenn Pointer oder Referenz verwendet wird, aber keine Basisklasse
inlining möglich	inlining unmöglich
is-a (LSP; provide no less demand no more)	has-a

## public, protected or private data?

- public ist nur sinnvoll, wenn es sich um eine lose Ansammlung von Daten handelt (struct; z.B. `std::pair`).
- private ist überall sonst zu verwenden.
- protected sollte überhaupt nicht verwendet werden (selbst die Person, die es in den Standard hinein argumentiert hat, ist in der Zwischenzeit anderer Meinung).
- Dies gilt für Daten (Instanzvariablen)! Für Funktionen (Methoden) können alle drei Zugriffsbeschränkungen sinnvoll sein.

## Get your interface right

- Das Interface (public und eventuell auch private Methoden) einer Klasse lässt sich kaum mehr korrigieren, wenn die Klasse bereits in weiter Verwendung ist.
- Die Implementierung kann später viel leichter angepasst werden.
- Open/Closed Principle:
  - software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification (ursprüngliche Form)
  - Vererbung von abstrakter Basisklasse (Interface)

## Weitere oft zitierte Regeln

- Don't Repeat Yourself (Every piece of knowledge must have a single, unambiguous, authoritative representation within a system)
  - Law of Demeter (Principle of Least Knowledge)
  - Single Responsibility Principle (Only one reason to change)
  - Interface Segregation Principle (no client should be forced to depend on methods it does not use)
  - Dependency Inversion Principle (
    - A. High-level modules should not depend on low-level modules. Both should depend on abstractions.
    - B. Abstractions should not depend on details. Details should depend on abstractions)
  - SOLID (SRP, Open/Closed Principle, LSP, ISP, DIP)
-



## Das Klasseninterface in C++

- In C++ ist die Trennung zwischen Interface einer Klasse (Methoden, die für die Objekte der Klasse verfügbar sind) und Implementierung der Klasse nicht ganz sauber.
- Die Klassendefinition (meist in einer Header-Datei) enthält neben der Definition des Interfaces auch einen Teil der Implementierung (z.B. Instanzvariablen)
- Änderungen in der Implementierung einer Klasse führen somit dazu, dass alle Client-Klassen neu kompiliert werden müssen.

# Das pimpl Idiom

- Pointer to Implementation Idiom: kann verwendet werden, um die Kompilationsabhängigkeiten aufzubrechen.

```
// in header file
class widget {
public:
    widget();
    ~widget();
private:
    class impl;
    unique_ptr<impl> pimpl;
};
```

```
// in implementation file
class widget::impl {
    // :::
};
```

```
widget::widget() : pimpl{ new impl{ /*...*/ } } { }
widget::~~widget() = default                // or {}
```

Konstruktor und Destruktor müssen notwendigerweise nach der Definition von `widget::impl` definiert werden, da bei `new` und Destruktor von `unique_ptr` die Objektgröße (von `widget::impl`) bekannt sein muss..



# Java

- Eine kurze Einführung

## **The Java® Language Specification Java SE 8 Edition**

**James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley**

**2015-02-13**

**<https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>**

# Fakten

	<b>C++</b>	<b>Java</b>
<b>Historie</b>	1983 Bjarne Stroustrup	1995 James Gosling
<b>basiert auf</b>	C, Simula, C with classes	- (Syntaxanleihen von C und C++)
<b>Paradigma</b>	imperativ, objektorientiert, streng typisiert	
<b>Standard</b>	INCITS/ISO/IEC 14882:2014 (2016)	de facto Standard (kontrolliert durch den Java Community Process JCP)
<b>aktuelle Version</b>	C++14 (C++17 in Arbeit)	Java 8 (Java 9 und Java 10 in Arbeit)
<b>Übersetzung</b>	Ahead of time (AOT) Compiler erzeugt Maschinencode	AOT Compiler erzeugt Bytecode, der von der Java Virtual Machine (JVM) interpretiert wird. Schon ab Version 1.2 mit Just in time (JIT) Compiler. Ab Version 9 zusätzlich AOT compiler für Maschinencode als Sprachbestandteil
<b>Ziele</b>	maximale Effizienz	simple, object-oriented, familiar robust and secure architecture neutral and portable (WORA – write once run anywhere) high performance interpreted, threaded and dynamic

# Organisation der Programmteile

- Die Dateien werden in Java in Packages organisiert.
- Um die Namen von Packages eindeutig zu halten, werden oft URLs genutzt, z.B.:  
at.ac.univie.cs.apps.ewms (die Umkehrung der Domain-Namen ist erforderlich, da URLs eigentlich von rechts nach links zu lesen sind).
- Jede öffentliche (public) erste Ebene (top level) Klasse muss in einer Datei gleichen Namens (mit Endung .java) definiert werden.
- Wir verwenden eine IDE (Eclipse), die uns diese Organisationsarbeit weitestgehend abnimmt. (IntelliJ wäre eine alternative IDE, die von vielen als überlegen angesehen wird.)
- (Anmerkung: Eine Java Umgebung kann auch z.B. eine Datenbank statt einzelner Files verwenden, um die Programmteile zu speichern und zu organisieren. Dann sind die Regeln nicht ganz so strikt.)

# Kompilieren und Programmstart

- Zunächst muss die Java Datei in JVM-Bytecode übersetzt werden (dabei werden .class-Dateien für alle enthaltenen Klassendefinitionen erzeugt):
    - `javac MyTest.java`
  - Gestartet wird das Programm in der JVM
    - `java MyTest`
  - Debuggen kann man mit dem Java Debugger
    - `jdb MyTest`
  - Damit das funktioniert, muss Java ordnungsgemäß installiert sein. Insbesondere muss die Umgebungsvariable CLASSPATH korrekt gesetzt sein.
  - Zum Debuggen empfiehlt sich Übersetzen mit der Option `-g`. Die Option `-Djava.compiler=NONE` weist den Debugger an, in der JVM keinen JIT-Compiler einzusetzen.
  - Auch hier erleichtert uns die IDE die Arbeit.
-

# Hello world

```
package hello.world;

public class Hello {

    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

- Wie in C++ startet das Programm mit main. Allerdings müssen alle Funktionen in Klassen gekapselt werden, auch main.

## Namenskonventionen in Java (naming conventions)

- Klassen, Interfaces mit großem Anfangsbuchstaben
- Primitive Datentypen, Methoden, Variablennamen mit kleinem Anfangsbuchstaben
- CamelCase für zusammengesetzte Wörter (`MeineKlasse`, `meineMethode`)
- Paketnamen nur Kleinbuchstaben (`meinedomain.meinepakete.paket1`)
- Konstante nur Großbuchstaben (`MEIN_KONST`)



# Primitive Datentypen (primitive datatypes)

ausgeschrieben

Exakte Größe im Speicher nicht definiert

Nicht für lokale Variable

Typ	Werte	Größe	Default
boolean	true, false	1 Bit	false
char	Unicode Zeichen	2 Byte	\u0000
byte	Ganze Zahlen	1 Byte	(byte)0
short	Ganze Zahlen	2 Byte	(short)0
int	Ganze Zahlen	4 Byte	0
long	Ganze Zahlen	8 Byte	0L
float	"Reelle Zahlen"	4 Byte	0.0f
double	"Reelle Zahlen"	8 Byte	0.0

**Berechnungen  
korrekt modulo  
 $2^x$**

**Rechenfehler  
NEGATIVE\_INFINITY  
POSITIVE\_INFINITY  
NaN**

## Literale (literal)

- Literale sind konstante Werte, die "buchstäblich" im Programmtext auftreten.
- Boolesche Literale: **true**, **false**
- Ganzzahlige Literale:
  - **11**, **+7**, **-91**
  - **231**, **23L** (**int** ist default)
  - **012** (oktal)
  - **0xa1**, **0Xff** (hexadezimal)
  - **0b101**, **0B0101** (binär, seit J2SE 7.0)

## Literale (2)

- Gleitkommazahlen Literale:
  - `27.5`, `+0.`, `-.3`
  - `1.0d`, `7.D`, `.5f`, `0.12F` (`double` ist default)
  - `0.2e3`, `4E-2`, `0.1e+7` (technische Notation)
  - `0x1.8p2`, `0XA.P-2` (hexadezimal seit JDK 5)
  - `Double.POSITIVE_INFINITY`,  
`Float.NEGATIVE_INFINITY`, `Double.NaN`

Jede Klasse bildet einen  
eigenen **Namensbereich**  
(namespace)

## Literale (3)

- In numerischen Literalen (ganz oder Gleitkomma) kann zwischen zwei Ziffern ein `_` (underscore) zum Gruppieren verwendet werden (ab Java SE 7):

– `1_000_000`

Seit C++14 ist in C++  
dafür die Verwendung  
von `'` möglich

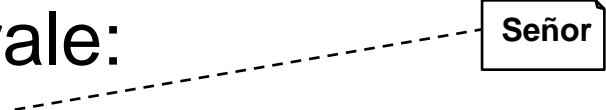
## Literale (4)

- Zeichen Literale:
  - `'a'`, `'0'` (beliebige Unicode UTF-16 Zeichen)
  - `'\u00F1'` (Unicode Zeichennummern dürfen auch innerhalb von Zeichenketten oder in Namen von z.B. Instanzvariablen verwendet werden. Hier: ñ)
  - `'\b'`, `'\t'`, `'\\'` (Escape-Sequenzen; escape sequence)

## Literale (5) Escape-Sequenzen

\n	newline	Zeilenvorschub
\t	tab	Tabulator
\b	backspace	Zeichen zurück
\r	carriage return	Zeilenbeginn
\f	formfeed	Seitenvorschub
\\	backslash	Umgekehrter Schrägstrich
\'	single quote	Einfaches Hochkomma
\"	double quote	Doppeltes Hochkomma
\dnnn	octal	Oktale Nummer
\xnn	hexadecimal	Hexadezimale Nummer
\unnnn	Unicode	Unicode Nummer

## Literale (6)

- Zeichenketten (String) Literale:
  - `"abc"`, `""`, `"Se\u00f1or"` 
  - Strings sind in Java konstante Objekte. Das heißt: sobald das Objekt einmal erzeugt wurde, lässt sich sein Inhalt nicht mehr verändern.

## Literale (7)

- Sonstige:
  - **null** (Leeres Objekt, dessen Datentyp auch **null** ist. Kompatibel mit allen Klassen)
  - **class** (Erzeugt ein Objekt für eine Klasse selbst. Z.B.: **Person.class**. Datentyp des Objekts ist **Class**. Auch für primitive Typen verwendbar; **boolean.class**, **void.class**)

Tony Hoare nennt seine Erfindung von **null** einen „billion-dollar mistake“.



## Variablen

- Variablen werden analog zu C++ definiert.
- Einfache Datentypen verhalten sich wie in C++.
- Variablen, die ein Objekt präsentieren, sind allerdings immer Referenzen!

```
int i = 3;  
int j = i;  
i = 5;  
System.out.println(i+", \n"+j);
```

**Ausgabe:**

5,  
3

```
Person a = new Person("A");  
Person b = a;  
a.setGewicht(100);  
System.out.println(a+", \n"+b);
```

**Ausgabe:**

A (0 Jahre, 40.0 cm, 100.0 kg),  
A (0 Jahre, 40.0 cm, 100.0 kg)

# Java Arrays

- Definition:  
`int[] arr;`
  - Instanziierung:  
`arr = new int[7];`
  - Instanziierung durch Initialisierung bei der Definition:  
`int[] arr = {1,2,3};`
  - Größe des Arrays ist fix.
  - Wird ein Array mit **new** erzeugt, so werden alle Elemente mit Defaultwerten initialisiert ('leer').
  - Arrayindizes beginnen mit 0.
  - Sind die Arrayelemente Objekte, so werden Referenzen (nicht Kopien) gespeichert.
-

# Operatoren und Ausdrücke

- Die verfügbaren Operatoren und Ausdrücke sind im Wesentlichen dieselben wie in C++ (zusätzlich gibt es noch >>> - right shift with 0 fill und Operatoren für Pointer -> \* & sowie scope Operator :: fehlen)
- Prioritäten und Assoziativitäten sind im Wesentlichen gleich.
- Allerdings ist die Reihenfolge der Abarbeitung von Operanden definiert mit linker Operand vor rechtem Operand und Seiteneffekte finden sofort statt. Es gibt daher keine undefinierten Ausdrücke
- ```
int i = 7;  
i += (i=4);  
System.out.println(i); //Gibt 11 aus
```

## Ausdrücke: Unterschiede zu C++

- Auf der linken Seite einer Zuweisung muss eine Variable stehen (keine lvalue-Ausdrücke)
- Keine implizite Typumwandlung zwischen boolean und int
- Keine einschränkende implizite Typumwandlung von double (float) auf int bei Zuweisung.
- Syntax für Typumwandlung ist die alte C/C++ Syntax mit dem gewünschten Datentyp in Klammern z.B.  
`int x=(int)1.5;`

## Operatoren können nicht überladen werden

- Die Bedeutung aller Operatoren ist fix vorgegeben. Einige können nur mit eingebauten primitiven Datentypen verwendet werden (+, -, etc.)
- Vorsicht bei Verwendung von `==`. Dieser Operator prüft auf Objektidentität (entspräche in C++ etwa einem Vergleich der Adressen). Für die meisten Datentypen ist daher ein Vergleich mit `.equals()` vorzuziehen. (Die Defaultimplementierung dieser Methode in `Object` fällt aber auf `==` zurück.)  
Vor allem wichtig beim Vergleich von Strings!

## Kontrollstrukturen

- Die aus C++ bekannten Kontrollstrukturen sind analog verwendbar (es gelten kleinere Einschränkungen, wie fehlende implizite Typumwandlung auf boolesche Werte und keine Möglichkeit im steuernden Ausdruck lokale Variable zu definieren)
- **if, while, do while, for, switch, break, continue, return**
- In switch Statements können seit Java 7 auch Strings zur Fallauswahl verwendet werden.

# Exceptions

- Werfen einer Exception mittels throw-Statement.
- Vergleichbar mit einem "Hilferuf"
- Nur Objekte, die (direkt oder indirekt) der Klasse **Throwable** angehören, können geworfen werden.

Unterschied zu C++!

```
throw new Throwable();
```

- Durch das Werfen einer Exception wird der normale Programmfluss unterbrochen und es wird bei einer passenden Fehlerbehandlungsroutine fortgesetzt.

# try, catch, finally

```
try {  
    //Block in dem eine Exception entstehen kann (auch indirekt durch einen Funktionsaufruf)  
}  
catch (ExceptionKlasse1 e) {  
    //Block zur Behandlung von Fehlern der ExceptionKlasse1  
}  
catch (ExceptionKlasse2 e) {  
    //Block zur Behandlung von Fehlern der ExceptionKlasse2  
}  
...  
finally {  
    //Block der auf jeden Fall am Ende ausgeführt wird (Exception oder nicht)!  
    //Einzigste Ausnahme: Programmabbruch (z.B. mittels exit())  
}
```

- Es muss zumindest ein Block (catch oder finally) nach dem try-Block kommen.
- Höchstens ein finally-Block ist erlaubt.

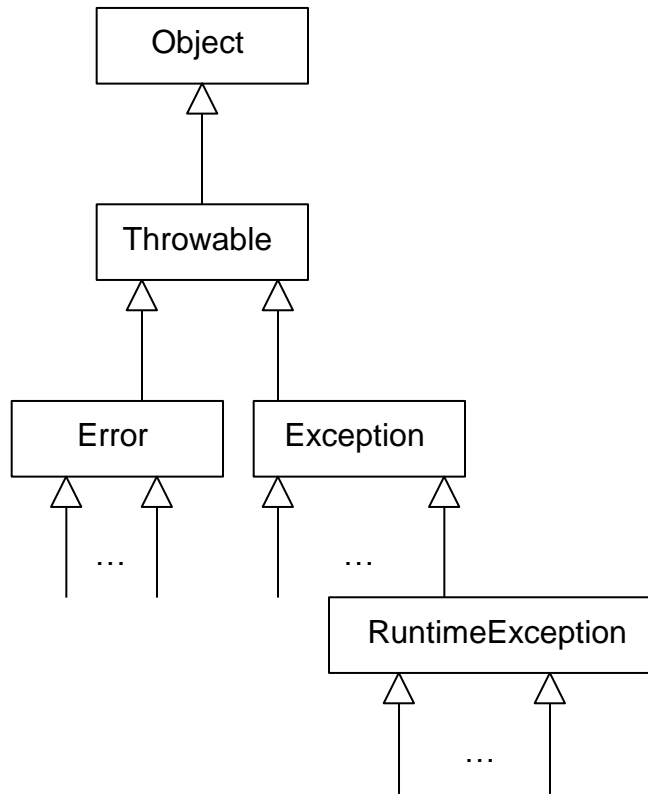


## try with resources

```
try (FileInputStream fis = new FileInputStream("filename")) {  
    //... Verarbeitung der Daten  
}  
catch (IOException e) {  
    //... Exception behandeln  
}
```

- Beliebig viele Ressourcen durch Semikolon getrennt anführbar
- Jede Ressource wird garantiert mit `close` geschlossen (Ressourcen müssen `AutoCloseable` oder davon abgeleitetes Interface implementieren)
- Eine Exception aus dem `try`-Block unterdrückt eventuelle Exceptions bei den `close` Operationen
- Alle Ressourcen werden geschlossen, bevor `catch`- oder `finally`-Blöcke exekutiert werden
- Eventuell unterdrückte Exceptions können mittels der Methode `getSuppressed()` aus der propagierten Exception ausgelesen werden
- Ab Java Version 7 verfügbar

# Exceptions Klassenhierarchie



- Die catch-Blöcke werden in der Definitionsreihenfolge nach Treffern durchsucht.
- Implizite Typumwandlungen in die Basisklasse werden durchgeführt.
- Die Reihenfolge der catch-Blöcke ist daher wichtig. (Zuerst die spezifischeren Klassen, später die weniger spezifischen.)

## catch or specify

- In Java gilt zunächst das Prinzip "catch or specify". Das heißt: Eine Funktion muss entweder dafür sorgen, dass eine Exception nicht nach außen dringt (indem sie abgefangen und behandelt wird), oder durch eine **throws**-Klausel spezifizieren, dass sie eventuell diese Exception werfen könnte.

```
public static void test() throws PrinterException, IOException {  
    try {  
        //Hier können unter Umständen PrinterException, IOException oder  
        //ClassNotFoundException geworfen werden  
    }  
    catch(ClassNotFoundException cnfe) {  
        //Fehlerbehandlung  
    }  
    finally {  
        System.out.println("Dies wird immer ausgegeben");  
    }  
}
```

## checked und unchecked Exceptions

- Es gibt Exceptions (z.B. **NullPointerException**) die prinzipiell fast überall auftreten können. Würde man erzwingen, dass alle diese Exceptions immer aufgelistet oder gefangen werden, wäre das ein ziemlicher Overhead. Daher gibt es die unchecked Exceptions (das sind **RuntimeException** und **Error**, sowie alle, die davon erben). Diese dürfen im Programm einfach ignoriert werden.
- Alle anderen sind checked Exceptions und für sie gilt: catch or specify.

## assertions

- Eine assertion ist eine Korrektheitsbedingung, die an beliebiger Stelle im Programm eingefügt werden kann.
- Zur Laufzeit wird geprüft, ob diese Bedingung an dieser Stelle erfüllt ist. Ist das nicht der Fall, wird das Programm abgebrochen.

```
assert expression;
```

```
assert expression: string;
```

## Aktivieren/Deaktivieren von assertions

- Assertions sind im Prinzip nur beim Testen von Programmen sinnvoll. Zur Laufzeit im Echtbetrieb bedeuten sie nur zusätzlichen Aufwand.
- Man kann die Überprüfung von Assertions daher ausschalten.
- Dies geschieht mit der Angabe von Schaltern beim Aufruf der JVM (Java Virtual Machine)
- -ea: enable assertions
- -da: disable assertions (default, muss nicht angegeben werden)

```
java -ea meinProgramm
```

## Globale Funktionen

- In Java gibt es keine globalen Funktionen. Funktionen werden in Utility Klassen zusammengefasst und als statische Methoden (Klassenmethoden) implementiert.
- Die Verwendung von Utility Klassen ist in der Objektorientierten Programmierung durchaus umstritten.
- Beispiele: Math, Files, Random
- Auch main muss in einer Klasse implementiert werden

# Parameterübergabe

- Alle Parameter werden in Java prinzipiell by value übergeben.
- Die oft anzutreffende Behauptung, Objekte würden by reference übergeben, beruht auf einem Missverständnis und darauf, dass Objektvariablen immer Referenzen sind.



## Überladen von Funktionen

- Regeln sind analog zu C++: Funktionen gleichen Namens mit unterschiedlicher Signatur überladen einander.
- Der Returntyp ist als Unterscheidungskriterium allein nicht ausreichend.
- Im Unterschied zu C++ werden in Java Funktionen gleichen Namens der Basisklasse überladen und nicht versteckt (in C++ ist dafür eine using-Deklaration nötig).

# Klassen

- Eine Klasse wird ähnlich wie in C++ definiert und enthält Instanzvariablen und Methoden (members)
- Anders als in C++ müssen aber die Sichtbarkeiten (access modifier) für jedes Element separat angegeben werden.

```
public class Test {  
    private int i;  
    public void method(int j) {}  
}
```

| Sichtbarkeiten |           |
|----------------|-----------|
| UML            | Java      |
| -              | private   |
| +              | public    |
| #              | protected |
| ~              |           |

In Java können nicht nur erbende Klassen zugreifen, sondern auch alle im selben Package

Default  
(package)

## Modifier static (analog zu C++)

- static kann als modifier für Variable und Methoden verwendet werden.
- statische Variable/Methoden werden als Klassenvariable bzw. Klassenmethoden bezeichnet.
- Klassenvariable existieren nur einmal für die gesamte Klasse, sie werden von allen Objekten gemeinsam benutzt.
- Klassenmethoden können ohne Objekte aufgerufen werden. In Klassenmethoden ist daher der Zugriff auf Instanzvariable und auf this verboten (Klassenvariable können verwendet werden).

## Modifier static (Beispiele)

```
public static void main() {  
    ...  
}
```

```
public class A {  
    private static int attr;  
    private static final int constant = 100;  
}
```

## Modifier static (Anwendungen)

- Klassenvariable
  - (Konstante) Vorgaben für alle Objekte einer Klasse
  - Laufende Nummern, Zählen existierender Objekte
  - Globale Steuerung des Verhaltens von Objekten
  - "Kommunikation" zwischen Objekten (sollte eher vermieden werden)
- Klassenmethoden
  - main
  - Methoden, die nicht auf Objekten operieren, aber doch eigentlich zur Klasse gehören (z.B.: erstellen einer temporären Datei in der Klasse File).
  - Utility-Funktionen (z.B. Klasse Math)

## Modifizier final (ähnlich zu C++ const bzw. final ab C++11)

- final kann als modifizier für Variable, Methoden, Parameter und Klassen verwendet werden.
- finale Variable sind konstante Werte. Sie müssen bei der Definition oder in jedem Konstruktor der Klasse mit einem Wert belegt werden.
- finale Methoden können nicht übersteuert werden (override ist verboten).
- finale Parameter können in der Funktion nicht verändert werden.
- finale Klassen können nicht als Basisklassen für Vererbung herangezogen werden.

## Modifier final (Beispiele)

```
private static final int MAXSIZE1 = 10;
private final int MAXSIZE2 = 10;
private final int MAXSIZE3;
public final void doNotOverride() {
    ...
}

public void noChanges(final int n, final Person p) {
    ...
}

public final class YouCannotDerive {
    ...
}
```

# Modifier final (Anwendungen)

- Konstanten
  - (Klassenweite) Vorgaben für Objekte (Größen von Arrays, etc.).
  - Abbildung von konstanten Werten aus der Realität (Pi).
  - Zwang, Konstante zu initialisieren hilft null Werte zu vermeiden.
  - Allgemein: Vermeidung von Literalen ("magic values") in Programmen.
- Methoden
  - Methoden, deren Übersteuern die Sicherheit der Klasse gefährden würde (speziell Methoden, die von Konstruktoren aufgerufen werden, sollten nicht übersteuert werden).
- Parameter
  - Sicherheit (Datenintegrität)
- Klassen
  - Hauptsächlich zur Gewährleistung von Sicherheit (z.B.: String Klasse).



## Modifier abstract

- abstract kann als modifier für Methoden und Klassen verwendet werden.
- abstrakte Methoden haben keine Implementierung. Sie dürfen aber aufgerufen werden. Der late binding Mechanismus sorgt dafür, dass die korrekte Implementierung aus einer der erbenden Klassen gewählt wird.
- abstrakte Klassen können nicht instanziiert werden, d.h. der Aufruf der Konstruktoren mit new ist verboten (Aufruf mittels super() ist weiterhin möglich).
- Eine Klasse, die eine abstrakte Methode enthält, muss selbst auch als abstrakt definiert werden.

## Modifier abstract (Beispiele)

```
public abstract void einkaufen();  
  
public abstract class A {  
    ...  
}
```

## Modifier abstract (Anwendungen)

- Methoden
  - Wenn jedes Objekt der Klasse zwar die Funktionalität anbietet, aber die Objekte sich je nach Unterklasse unterschiedlich verhalten und es nicht möglich ist, ein gemeinsames Grundverhalten für alle Objekte zu definieren.
- Klassen
  - Falls eine Methode abstrakt ist (nicht automatisch wie bei C++).
  - Falls die Instanziierung der Klasse logisch nicht sinnvoll erscheint (Zwang, eine bestimmte Unterklasse – einen bestimmten Typ – auszuwählen).

# Konstrukturen

- Analog zu C++ (Name der Klasse, kein Returnwert, Defaultkonstruktor wird automatisch generiert, wenn kein Konstruktor definiert wird, kann beliebig überladen werden).
- Aufruf von Konstruktoren (Erzeugen von Objekten) immer mit new (Achtung: großer Unterschied zu C++!)

```
new Person() //Aufruf Defaultkonstruktor  
new Person("Graf Bobby") //Person(String)
```

# Konstruktoren Verkettung (constructor chaining)

```
public Person(String name) {  
    this(name, new Date(), 'w', MIN_GROESSE, MIN_GEWICHT);  
}  
  
public Person(String name, Date geburtsdatum, char geschlecht,  
double groesse, double gewicht) {  
    this.name = name;  
    this.geburtsdatum = geburtsdatum;  
    this.geschlecht = geschlecht;  
    this.groesse = groesse;  
    this.gewicht= gewicht;  
}
```

- Verkettung kann über beliebig viele Konstruktoren laufen.
- Vermeidet Codevervielfältigung.
- Aufruf eines anderen Konstruktors mit `this()` muss immer allererstes Statement in einem Konstruktor sein.

## Destruktor

- Wird aufgerufen, wenn ein Objekt zerstört wird.
- Dient zum "Aufräumen" (z.B. Freigabe von Ressourcen).
- In Java realisiert durch die Methode **finalize()** (in der Klasse **Object** vordefiniert, kann overridden werden).
- In Java ist nicht definiert, dass auch für jedes Objekt am Ende des Lebenszyklus **finalize()** aufgerufen wird (garbage collector). Daher nur beschränkt einsetzbar.
- Destruktor hat in der Regel keine Parameter und kann daher auch nicht überladen werden.

## Simple I/O an der Konsole

- `System.out.println()` gibt Strings aus (Objekte werden durch Aufruf der Methode `toString` bei Bedarf in Strings konvertiert).
- Scanner kann zum Einlesen verwendet werden z.B.:

```
Scanner s = new Scanner(System.in) ;  
int i = s.nextInt() ;  
String s = s.next() ;  
s.close() ;
```

# Java Kommentare

- Analog zu C++
  - `//` startet ein Kommentar bis zum Ende der Zeile
  - `/*` startet ein Kommentar, das mit `*/` beendet wird
- Zusätzlich:
  - `/**` startet ein Kommentar, das von Javadoc verarbeitet wird



## Java Keyword `this`

- Bezeichnet in einer Methode das aktuelle Objekt (das Objekt, für welches die Methode aufgerufen wurde).
- Im Unterschied zu C++ ist `this` in Java eine Referenz und kein Pointer. Es wird also der Operator `.` statt `->` für den Zugriff auf Instanzvariablen/Methoden des Objekts verwendet.
- Analog zu C++ wird `this` implizit vom Compiler für den Zugriff auf Instanzvariablen/Methoden verwendet und muss in diesen Fällen nur angegeben werden, wenn Mehrdeutigkeiten (z.B. mit Parameternamen) entstehen.

## Java Enumerationen (1)

- Definition einer Enumeration in Java:

```
public enum Augenfarbe {BLAU, GRUEN, BRAUN, GRAU}
```

- entspricht in etwa:

```
public class Augenfarbe {  
    private Augenfarbe() {}  
    public static final BLAU = new Augenfarbe();  
    public static final GRUEN = new Augenfarbe();  
    public static final BRAUN = new Augenfarbe();  
    public static final GRAU = new Augenfarbe();  
}
```

privater Konstruktor verhindert, dass weitere Objekte vom Typ Augenfarbe erzeugt werden können.

Großbuchstaben, da es sich um konstante Objekte handelt. Da jedes Objekt unveränderlich und einzigartig ist, reichen Vergleiche mit `==` aus.

## Java Enumerationen (2)

- Vorteile von Enumerationen in Java
  - Typsicherheit
  - Namensraum (Augenfarbe.BLAU)
  - Kann auch innerhalb anderer Klassen vereinbart werden (Person.Augenfarbe.BLAU)
  - Kann mit switch-Anweisung verwendet werden
  - Da es sich prinzipiell um Klassenobjekte handelt, können enums auch um Instanzvariable und Methoden erweitert werden
  - Vordefinierte Methoden
    - `String name()`: retourniert den Namen des Objekts als String (z.B.: "BLAU")
    - `int ordinal()`: Ordinalzahl des Objekts (Nummerierung von 0 beginnend)
    - `String toString()`: wie `name()`, kann aber überladen werden, um benutzerfreundlichere Namen zu verwenden
    - `static E valueOf(String s)`: Liefert das Objekt namens s; `IllegalArgumentException` wird geworfen, falls das Objekt nicht existiert
    - `static E[] values()`: Liefert ein Array mit allen Enumelementen in der Reihenfolge ihrer Definition.

statische Methoden werden für die Klasse, nicht für ein Objekt aufgerufen

## Java Enumerationen (3)

- Weil Enumerationen prinzipiell wie Klassen sind, können auch zusätzliche Instanzvariablen und Methoden definiert werden. (Diese erweiterten Möglichkeiten werden in UML und C++ nicht angeboten!)
- Beispiel: jede Augenfarbe hat eine Häufigkeit

```
public enum Augenfarbe {BLAU(46), GRUEN(26), BRAUN(16), GRAU(12);  
    public final int PROZ;  
  
    private Augenfarbe(int proz) {  
        this.PROZ=proz;  
    }  
}
```

Konstruktor muss privat sein!

Kein Defaultkonstruktor mehr.  
Parameter muss angegeben werden!

Strichpunkt nun nicht mehr optional.

## Java Enumerationen (4)

- Erweiterungen mit Methoden und eigenem Override für spezifische Konstanten:

```
public enum Augenfarbe {  
    BLAU(),  
    GRUEN(26) {  
        @Override  
        public String toString() {return "Cool";}  
    },  
    BRAUN(16),  
    GRAU(12);  
  
    public final int PROZ;  
  
    private Augenfarbe() {this.PROZ=46;}  
    private Augenfarbe(int proz) {this.PROZ=proz;}  
  
    @Override  
    public String toString() {return name();}  
}
```

## Java Enumerationen (5)

- Die definierten Konstanten werden immer mit dem Namen der Enumeration qualifiziert, z.B. **Augenfarbe.BLAU** (vgl. `scoped enumeration` in C++).

# Vererbung in Java (1)

```
class Delphin extends Saeugetier {
```

```
    @Override
```

Annotation

→Annotations

```
    void atmen() {
```

```
        ...
```

```
        super.atmen();
```

Aufruf der Methode in  
der Basisklasse

```
        ...
```

```
    }
```

```
    ...
```

```
}
```

- Wird keine extends Klausel angegeben, so wird die Klasse Object implizit als Basisklasse verwendet, die somit die Basis für alle Klassen der gesamten Java Klassenhierarchie bildet.
- Es kann nur von einer Klasse geerbt werden (single inheritance)

## Vererbung in Java (2)

- Die abgeleitete Klasse erbt alle zugreifbaren (nicht private definierten) Instanzvariablen und Methoden.
- Wird in der abgeleiteten Klasse der Name einer ererbten Instanzvariable/Klassenvariable zur Definition einer anderen Variablen verwendet, so wird die Variable der Basisklasse versteckt (hiding) und ist nicht mehr direkt zugreifbar (möglicher Zugriff mittels Qualifizierung für statische Variable bzw. durch Verwendung des keywords super für Instanzvariable).
- Der gleiche Effekt tritt bei Verwendung des gleichen Namens in statisch verschachtelten Blöcken ein. Die Java Language Reference spricht in diesem Zusammenhang (um den Unterschied zur Vererbung zu betonen) von verdeckten Variablen (shadowing).



## Vererbung in Java (3)

- Wird eine Methode gleichen Namens in der abgeleiteten Klasse definiert, dann gibt es folgende Möglichkeiten:
  - Die beiden Signaturen (Anzahl und Typen der Parameter) sind gleich, dann wird die Methode überlagert (override), wenn die Methode in der abgeleiteten Klasse keine Klassenmethode (nicht static) ist, ansonst wird die Methode der Basisklasse versteckt (hidden). Falls der return Typ der beiden Methoden nicht gleich ist, dann muss der return Typ der Methode in der abgeleiteten Klasse ein Subtyp des return Typs der Methode in der Basisklasse sein (wegen der Substituierbarkeitsregel).
  - Die beiden Signaturen sind verschieden, dann wird die Methode überladen (overload).
- (Anmerkung: In der Java Language Reference müssen die beiden Signaturen nicht genau gleich sein, um ein Override zu erhalten. Es gelten spezielle Regeln für Generics-Parameter, die aber im Rahmen dieser LV nicht so genau behandelt werden.)

## Die `toString` Methode

- ist in der Java-Basisklasse **Object** definiert, von der alle Klassen (direkt oder indirekt) erben.
- wird von der Methode `println()` der Instanzvariable `out` (Objekt der Klasse **PrintStream**) der Klasse **System** verwendet, um ein Objekt in einen String "umzuwandeln".
- In der Klasse **Object** wird ein String im Format **Klassenname@Hashwert** erzeugt, z.B.: `version8.Person@65a77f`
- Es empfiehlt sich, die `toString` Methode für die eigenen Klassen zu override.
- Analog zu Überladen von `operator<<` für Klassen in C++

## Das Schlüsselwort **super** (1)

- Das keyword **super** erlaubt den Zugriff auf versteckte (hidden) und überlagerte (overridden) Instanzvariablen/Methoden.

```
super.instanzvariable=0;
```

```
super.methode ();
```

## Das Schlüsselwort **super** (2)

- In der Schreibweise als Methode kann von einem Konstruktor aus ein Konstruktor der Basisklasse aufgerufen werden.

```
Klasse(int i) {  
    super(i*2);    //notwendig, falls es keinen  
                  //Defaultkonstruktor gibt  
}
```

- **super()** muss das allererste Statement in einem Konstruktor sein. Als Parameter dürfen keine Instanzvariable verwendet werden (da diese noch nicht initialisiert sind).
- Wird im Konstruktor kein Konstruktor der Basisklasse explizit aufgerufen, dann wird automatisch ein Aufruf von **super()** (Defaultkonstruktor der Basisklasse) am Beginn der Konstruktormethode eingefügt. Hat die Basisklasse keinen Defaultkonstruktor, so wird ein Compilerfehler erzeugt.

# Interfaces

- Interfaces sind eine Ansammlung von Methoden und haben keinen internen Zustand
- Es können daher problemlos beliebig viele Interfaces in einer Klasse implementiert werden. Dadurch ist die Einschränkung der single inheritance nicht so gravierend.
- Interfaces können ebenfalls von anderen Interfaces erben.
- Variablen können den Datentyp eines Interfaces haben und damit jedes Objekt referenzieren, das dieses Interface anbietet.

# Interfacedefinition in Java (1)

```
public interface IExample {
```

```
    int j=3;
    int i=j+5;
```

```
    void method1();
    int method2(int i);
    default void method3() {
    }
    static void method4() {
    }
}
```

Attribute sind implizit **public static final**. Es ist möglich, dies bei der Definition entsprechend explizit zu schreiben, die Language Specification rät davon aber ab.

Attribute müssen auf jeden Fall mit einem Ausdruck initialisiert werden. Dabei dürfen andere Attribute des Interfaces, die später im Text definiert werden, nicht verwendet werden ( $j=3+i$  wäre also illegal).

Methoden sind implizit **public abstract**. Es ist möglich, dies bei der Definition entsprechend explizit zu schreiben, die Language Specification rät davon aber ab.

Methoden konnten keinesfalls **static** sein (weil sich das mit **abstract** nicht verträgt). Seit Java 8 gibt es die Möglichkeit Defaultimplementierungen für Interface-Methoden anzugeben und diese auch statisch zu definieren.

Jede Klasse, die das Interface implementiert, muss alle deklarierten Methoden, für die es keine Defaultimplementierung gibt, definieren.

Ein Interface kann auch andere Interfaces und Klassen verschachtelt (nested) enthalten. Dies wird aber in dieser LV nicht behandelt und ist deshalb in diesem Beispiel nicht dargestellt.

**<<interface>>**  
**IExample**

+j:int = 3 {ReadOnly}  
+i:int = j+5 {ReadOnly}

+method1()  
+method2(i:int):int

## Interfacedefinition in Java (2)

```
public interface Outputable {  
    String toString();  
}
```

Nicht abstrakte Methoden, die von einer Superklasse geerbt werden (hier toString aus Object) müssen in der implementierenden Klasse nicht extra implementiert werden, da sie ja schon vorhanden sind.

# Das Interface Comparable (1)

```
public interface Comparable {  
    int compareTo(Object o);  
}
```

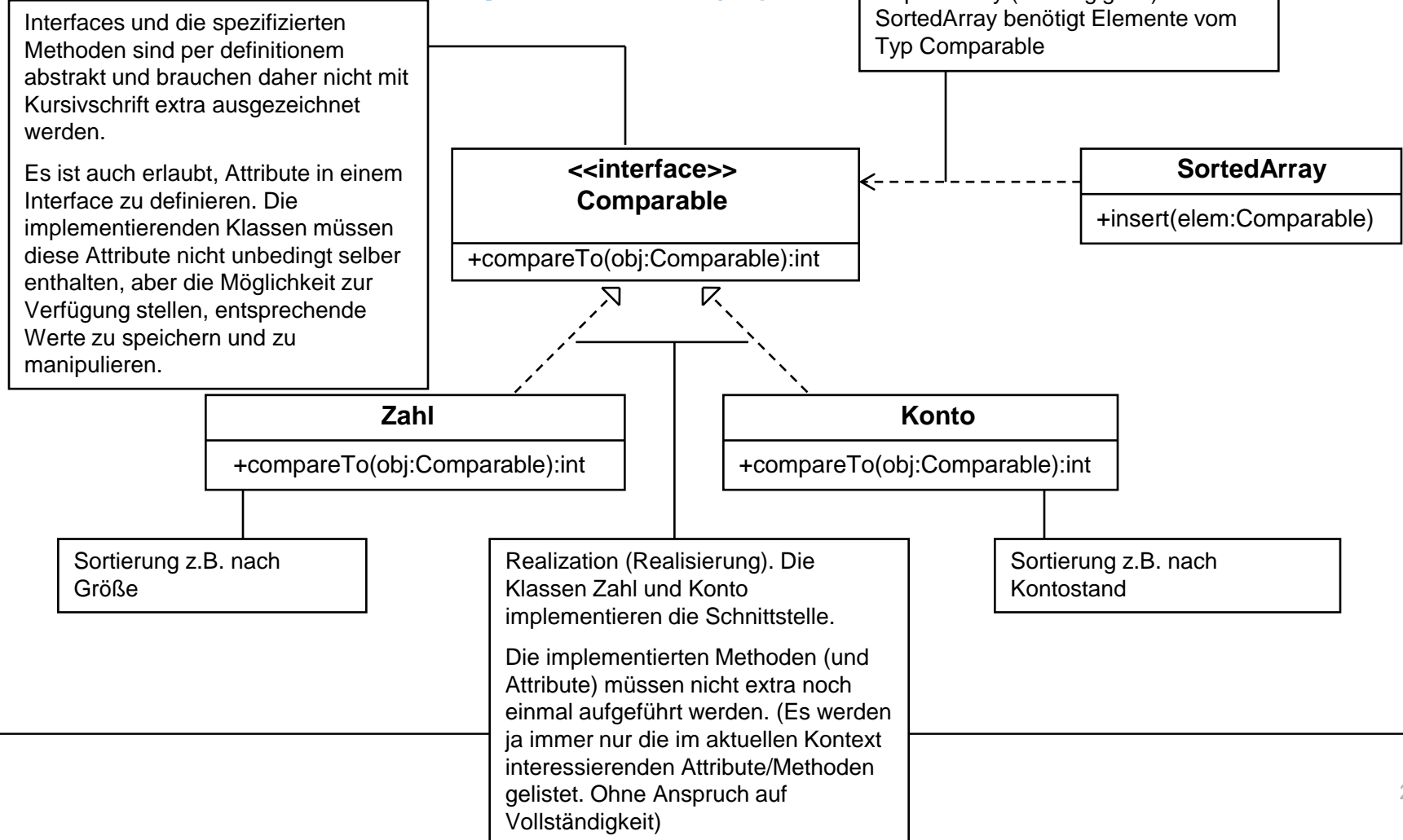
Dieses Interface ist im Java Framework vordefiniert. Will man es implementieren, so muss die Methode **compareTo** einen Wert kleiner Null retournieren, wenn das aktuelle Objekt (**this**) kleiner als das zu vergleichende (o) ist. Null, wenn beide Objekte gleich sind und einen positiven Wert sonst.

Ist das Objekt o nicht mit dem aktuellen Objekt vergleichbar (z.B. unpassende Datentypen), dann muss eine **ClassCastException** geworfen werden.

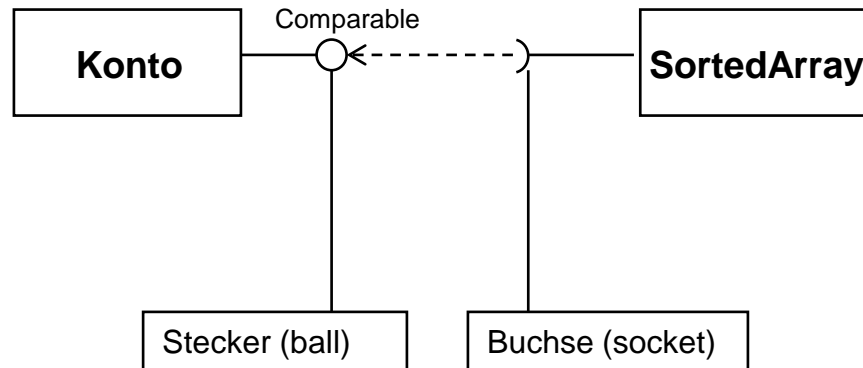
- Ein Interface ist ein Vertrag zwischen dem Anbieter (Realizator, Implementor) und dem Verwender (User). Dieser Vertrag umfasst nicht nur die anzubietenden Funktionen, sondern auch, was diese genau zu leisten haben. Diese Information findet man üblicherweise in der Dokumentation. Wie diese Leistung dann erbracht wird, bleibt der Implementation überlassen



## Das Interface Comparable (2)



# Kurzform Lollipop Notation



Ein Objekt kann beliebig viele Schnittstellen bereitstellen/nutzen:



## Beispiele „Vertragskonditionen“ (1)

- **add** Methode im Interface Collection: garantiert, dass nach dem Aufruf das einzufügende Element in der Collection enthalten ist. Daher muss eine Exception geworfen werden, falls ein Element nicht eingefügt werden kann, weil z.B. zu wenig Speicher vorhanden ist. Der Retourwert **false** ist reserviert für den Fall, dass das Element schon enthalten ist und nicht mehrmals eingefügt werden darf (z.B. **Set**).

## Beispiele „Vertragskonditionen“ (2)

- Die Methoden `compareTo` und `equals` müssen verträglich sein. Wenn `compareTo` für zwei Objekte 0 liefert, dann muss `equals` `true` liefern und vice versa.
- Eventuell muss dafür die Methode `equals` übersteuert (overridden) werden. Achtung: `equals` muss auch mit der Methode `hashCode` verträglich gehalten werden! Diese muss dann in der Regel auch übersteuert werden
- Nichteinhalten dieser Vertragskonditionen führt z.B. bei der Verwendung des Java Collection Frameworks zu allerlei unerwünschten und überraschenden Effekten, da sich das Framework blind auf die Gültigkeit der geforderten Übereinstimmungen verlässt

## Exkurs `equals` und `hashCode` (1)

Anmerkung:  
Mathematiker nennen  
eine reflexive,  
symmetrische und  
transitive Relation  
Äquivalenzrelation

- `equals` muss folgende Eigenschaften haben:
  - Reflexivität: `x.equals(x) == true`
  - Symmetrie: `x.equals(y) == y.equals(x)`
  - Transitivität: `x.equals(y) ∧ y.equals(z) → x.equals(z)`
  - `x.equals(null) == false` für alle Objekte
  - Konsistente (gleichbleibende) Ergebnisse solange sich nichts Relevantes an `x` und/oder `y` ändert
- `hashCode` muss folgende Eigenschaften haben
  - Liefert während des gesamten Laufs einer Applikation konsistent immer den gleichen Wert für ein Objekt, soweit sich keine der für `equals` relevanten Merkmale geändert haben.
  - `x.equals(y) → hashCode(x) == hashCode(y)`

Umgekehrte Richtung  
muss nicht gelten!

Hashfunktion in der  
Regel nicht injektiv.

## Exkurs `equals` und `hashCode` (2)

- Kompliziert?
- Eclipse:
- Source – Generate `hashCode` and `equals` ...

## Zurück zu Comparable

```
public interface Comparable {  
    int compareTo(Object o);  
}
```

|                                                 |
|-------------------------------------------------|
| <b>&lt;&lt;interface&gt;&gt;<br/>Comparable</b> |
| <b>+compareTo(o:Object):int</b>                 |

# Implementierung von Comparable

```
public class Konto implements Comparable {
    ...
    public int compareTo (Object other) {
        //nur Konten vergleichen
        if (other.getClass() != getClass())
            throw new ClassCastException();
        return (int) (kontoStand - ((Konto) other).kontoStand);
    }
}
```

Beliebig viele Interfaces können hier durch Komma getrennt angeführt werden

Muss implementiert werden! (**public** notwendig, weil im Interface implizit diese Sichtbarkeit festgelegt wurde.)

Unterschied zu `instanceof`?

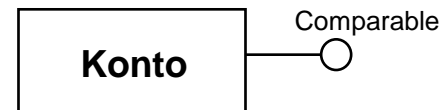
Typecasts (Typumwandlungen) und Klammerungen sind hier notwendig!

<<interface>>  
Comparable



Konto

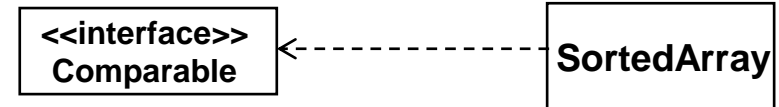
bzw.



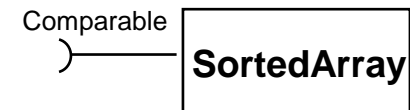


# Verwendung von Comparable

```
public class SortedArray {  
    Comparable[] store = new Comparable[100];  
    ...  
    public void add (Comparable elem) {  
        ...  
        if (store[i].compareTo(elem))  
        ...  
    }  
    ...  
}
```



bzw.



## Problem mit Comparable

- Wir können jetzt in einer Klasse **Comparable** implementieren und Objekte dieser Klasse dann z.B. im Java Collection Framework sortieren.
- Aber Objekte von Klassen, die wir nicht verändern können/wollen, können wir auf diese Art noch immer nicht sortieren

# Lösung

- Interface **Comparator**

```
public interface Comparator {  
    int compare(Object o1, Object o2);  
    boolean equals (Object obj);  
}
```

Muss bei der Implementierung nicht unbedingt programmiert werden, da schon in **Object** vorhanden

- zwei Objekte können nun "von außen" verglichen werden
- Comparator** kann nun z.B. beim Konstruktorauf Ruf eines **SortedSet** mitübergeben werden.

## Vererbung von Interfaces

- Syntaktisch wie bei Klassen. Unterschied: es kann von mehreren (beliebig vielen) Interfaces geerbt werden.

```
public interface iAll extends Comparable, Comparator {  
    ...  
}
```

## Anonymous Classes (1)

- Einfache Interfaces, die oft benötigt werden, aber meistens völlig unterschiedlich implementiert werden (wie z.B.: Comparable, Comparator, ActionListener) haben erheblichen Overhead.
- Es muss eine public Klasse definiert werden, von der ein entsprechendes Objekt erzeugt wird, das dann als Parameter oder direkt zum Aufruf der gewünschten Methode verwendet wird.
- Anonyme Klassen können den Aufwand verringern.

## Anonymous Classes (2)

- Eine anonyme Klasse wird in einem Ausdruck definiert, der gleich ein passendes Objekt der Klasse instantiiert, z.B.:

```
static void main(String[] args) {  
    SortedSet set = new TreeSet(new Comparator() {  
        @Override  
        public int compare(Object l, Object r) {  
            return l.toString().compareTo(r.toString());  
        }  
    });  
}
```

## Anonymous Classes Einschränkungen

- Es gelten dieselben Einschränkungen, wie für local classes (solche, die in einem Block definiert werden) allgemein.
- Zugriff auf die member der umschließenden Klasse ist erlaubt, auf lokale Variable aus dem umgebenden Gültigkeitsbereich darf nur zugegriffen werden, wenn diese **final** oder (ab Java 8) effectively final sind.
- Wie für alle nested classes wird durch die Deklaration eines Namens dieser Name im umgebenden Gültigkeitsraum (scope) eventuell überschattet (shadowing).
- Es ist nicht erlaubt, static initializers, member interfaces oder Konstruktoren zu deklarieren.

## Functional Interfaces

- Dieser Begriff wird in Java 8 für einfache Interfaces verwendet, die nur die Implementierung einer einzigen Methode verlangen (alle anderen, eventuell deklarierten Methoden müssen eine default-Implementierung haben oder ererbt werden).
- Statt anonyme Klassen für solche Interfaces zu schreiben, gibt es die Möglichkeit, die syntaktisch etwas einfacheren, in Java 8 neu hinzugekommenen Lambda Expressions zu verwenden.



# Lambda Expressions (Syntax)

**parameter -> expression body**

- Typangabe für die Parameter ist optional
- Runde Klammern um Parameterliste kann weggelassen werden, wenn genau ein Parameter spezifiziert wird.
- Geschwungene Klammern um den Body sind optional, wenn nur ein Statement angegeben wird.
- Schlüsselwort `return` optional, wenn der Body nur aus einem Ausdruck besteht.

```
static void main(String[] args) {  
    SortedSet set = new TreeSet((Object l, Object r) -> {  
        return l.toString().compareTo(r.toString());  
    });  
    SortedSet set = new TreeSet((l, r) ->  
        l.toString().compareTo(r.toString());  
    );  
}
```

# Java Generics

- Seit Java 5 Bestandteil der Sprache.
- Beheben die Schwierigkeit, Interfaces und Klassen typsicher zu implementieren (z.B. Comparable für Personen, oder eine Liste von Autos)
- Verwenden eine ähnliche Syntax wie C++-Templates, haben aber überhaupt nichts damit zu tun.
- Die Typinformationen werden vom Compiler entfernt, die Korrektheit wird nur statisch zur compile time überprüft. Zur Laufzeit wird mit den Grundtypen (raw types wie Comparable, List) bzw. mit passenden Basisklassen gearbeitet. Dieses Vorgehen wird als **type erasure** bezeichnet.
- Es herrschen typischerweise keine Vererbungsbeziehungen zwischen „verwandten“ Generics, wie z.B. List<Person> und List<Object>
- Wir betrachten Generics in dieser LV weitgehend nur von der Anwendungsseite.

# Implementierung von Comparable mit Generics

```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

```
public class Konto implements Comparable<Konto> {  
    ...  
    public int compareTo (Konto other) {  
        return (int)(kontoStand - other.kontoStand);  
    }  
}
```

Typüberprüfungen und –umwandlungen  
nicht mehr notwendig.

## Implementierung eines generischen Typs, der Comparable verwendet

```
public class SortedArray<T extends Comparable<? super T>> {  
    T[] store;  
    int size;  
  
    public SortedArray(T elem, int size){  
        @SuppressWarnings("unchecked")  
        T[] newArray = (T[])Array.newInstance(elem.getClass(), size);  
        store = newArray;  
        this.size = 0;  
    }  
    public void add (T elem) {  
        int i=size;  
        if (i>=store.length)  
            throw new IndexOutOfBoundsException("Maximale Elementanzahl überschritten!");  
        while (i>0 && store[i-1].compareTo(elem)>0) {  
            store[i]=store[i-1];  
            --i;  
        }  
        store[i]=elem;  
        ++size;  
    }  
    ...  
}
```

Generische und typsichere Implementierung  
nicht ganz trivial.

# Implementierung von Comparator mit Generics

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    boolean equals (Object obj);  
}
```

```
public class MyComp implements Comparator<Konto> {  
    public int compare(Konto l, Konto r) {  
        return (int)(l.getKontoStand() - r.getKontoStand());  
    }  
}
```

# Java generische Collections

|                              | ArrayList                | Vector                           | LinkedList               |
|------------------------------|--------------------------|----------------------------------|--------------------------|
| Interne Datenstruktur        | Array                    | Array                            | doppelt verkettete Liste |
| synchronisiert (thread safe) | Nein                     | Ja                               | Nein                     |
| Anfangsgröße                 | 10;<br>Konstruktoraufruf | 10;<br>Konstruktoraufruf         | 0                        |
| Wachstum                     | nicht spezifiziert       | verdoppeln;<br>Konstruktoraufruf | Einzelnes Element        |
| Zugriff                      | $O(1)$                   | $O(1)$                           | $O(n)$                   |
| Einfügen, Löschen            | $O(1)$ , $O(n)$          | $O(1)$ , $O(n)$                  | $O(1)$                   |

konstanter amortisierter Aufwand für das Einfügen eines Elements

# Grundlegende Collection-Methoden

|                                         |                                                                                                                        |
|-----------------------------------------|------------------------------------------------------------------------------------------------------------------------|
| <code>boolean add(E elem)</code>        | Hinzufügen eines Elements, liefert gegebenenfalls (z.B. bei Mengen) <code>false</code> , falls Element schon enthalten |
| <code>void clear()</code>               | Collection leeren                                                                                                      |
| <code>boolean contains(Object o)</code> | Liefert <code>true</code> , falls Objekt enthalten ist                                                                 |
| <code>boolean isEmpty()</code>          | Liefert <code>true</code> für leere Collection                                                                         |
| <code>boolean remove(Object o)</code>   | Entfernen eines Elements                                                                                               |
| <code>int size()</code>                 | Anzahl der gespeicherten Elemente                                                                                      |
| <code>Object[] toArray()</code>         | Liefert Array, das alle gespeicherten Elemente enthält                                                                 |

- Die Methoden `add`, `clear` und `remove` sind optional. Das heißt, sie müssen nicht von jeder Collectionklasse unterstützt werden. Sind sie nicht implementiert, so muss beim Aufruf eine Exception vom Typ `UnsupportedOperationException` geworfen werden.

# Das Interface Collection (1)

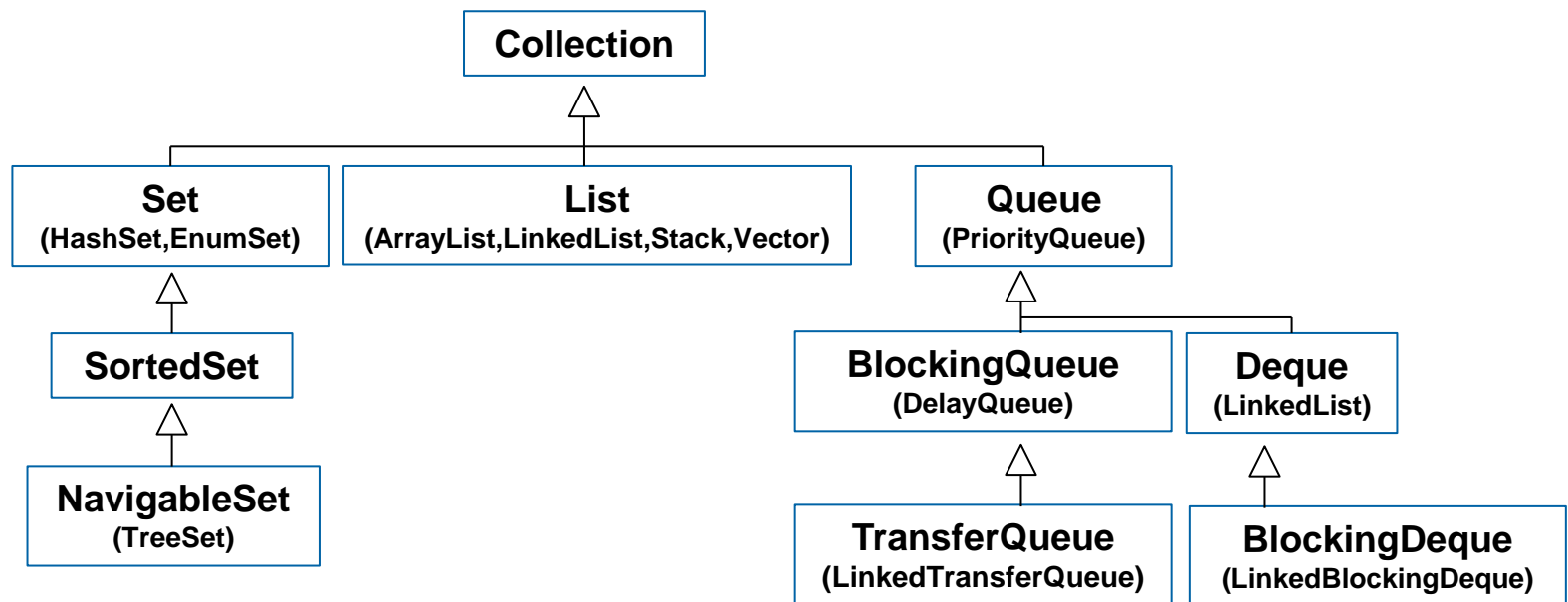
```
boolean add(E e)
    Ensures that this collection contains the specified element (optional operation).
boolean addAll(Collection<? extends E> c)
    Adds all of the elements in the specified collection to this collection (optional operation).
    void clear()
        Removes all of the elements from this collection (optional operation).
boolean contains(Object o)
    Returns true if this collection contains the specified element.
boolean containsAll(Collection<?> c)
    Returns true if this collection contains all of the elements in the specified collection.
boolean equals(Object o)
    Compares the specified object with this collection for equality.
    int hashCode()
        Returns the hash code value for this collection.
boolean isEmpty()
    Returns true if this collection contains no elements.
Iterator<E> iterator()
    Returns an iterator over the elements in this collection.
boolean remove(Object o)
    Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean removeAll(Collection<?> c)
    Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean retainAll(Collection<?> c)
    Retains only the elements in this collection that are contained in the specified collection (optional operation).
    int size()
        Returns the number of elements in this collection.
Object[] toArray()
    Returns an array containing all of the elements in this collection.
<T> T[] toArray(T[] a)
    Like toArray(); the runtime type of the returned array is that of the specified array.
```



## Das Interface `Collection` (2)

- Das Collection Framework bietet keine Klasse, die das Collection Interface direkt implementiert.
- Es gibt aber eine Reihe von Interfaces, die von Collection erben, mit entsprechenden konkreten Implementierungen.

# Collection Framework Interface Hierarchy



## Das Interface **Set**

- keine Erweiterung der Funktionalität von **Collection**
- zusätzliche Einschränkung: Elemente dürfen nicht doppelt eingetragen werden (Elemente sind gleich, wenn der Vergleich mittels **equals** true ergibt)
- konkrete Implementierung
  - **HashSet**

# Das Interface `SortedSet`

- Erweiterung von `Set`

`Comparator<? super E> comparator()`

Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.

`E first()`

Returns the first (lowest) element currently in this set.

`SortedSet<E> headSet(E toElement)`

Returns a view of the portion of this set whose elements are strictly less than toElement.

`E last()`

Returns the last (highest) element currently in this set.

`SortedSet<E> subSet(E fromElement, E toElement)`

Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.

`SortedSet<E> tailSet(E fromElement)`

Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

- konkrete Implementierung
  - `TreeSet`

## Die Klasse `EnumSet`

- Konkrete Implementierung des Interface `Set` für Enum-Datentypen.
- Keine (öffentlichen) Konstruktoren.
- Erstellen von `EnumSets` mit speziellen (statischen) Methoden.
- Danach können Sie wie andere Collections auch mit `add` und `remove` verändert werden.

## EnumSet erstellen (1)

|                                                                      |                                                 |                                                                                                                                                                           |
|----------------------------------------------------------------------|-------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>static &lt;E extends Enum&lt;E&gt;&gt; EnumSet&lt;E&gt;</code> | <code>allOf(Class&lt;E&gt; elementType)</code>  | Creates an enum set containing all of the elements in the specified element type                                                                                          |
| <code>EnumSet&lt;E&gt;</code>                                        | <code>clone()</code>                            | Returns a copy of this set                                                                                                                                                |
| <code>static &lt;E extends Enum&lt;E&gt;&gt; EnumSet&lt;E&gt;</code> | <code>complementOf(EnumSet&lt;E&gt; s)</code>   | Creates an enum set with the same element type as the specified enum set, initially containing all the elements of this type that are not contained in the specified set. |
| <code>static &lt;E extends Enum&lt;E&gt;&gt; EnumSet&lt;E&gt;</code> | <code>copyOf(Collection&lt;E&gt; c)</code>      | Creates an enum set initialized from the specified collection                                                                                                             |
| <code>static &lt;E extends Enum&lt;E&gt;&gt; EnumSet&lt;E&gt;</code> | <code>copyOf(EnumSet&lt;E&gt; s)</code>         | Creates an enum set with the same element type as the specified enum set, initially containing the same elements (if any)                                                 |
| <code>static &lt;E extends Enum&lt;E&gt;&gt; EnumSet&lt;E&gt;</code> | <code>noneOf(Class&lt;E&gt; elementType)</code> | Creates an empty enum set with the specified element type                                                                                                                 |

## EnumSet erstellen (2)

|                                                                                                        |                                                                                                                  |
|--------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| <code>static &lt;E extends Enum&lt;E&gt;&gt; EnumSet&lt;E&gt; of (E e)</code>                          | Creates an enum set initially containing the specified element                                                   |
| <code>static &lt;E extends Enum&lt;E&gt;&gt; EnumSet&lt;E&gt; of (E first, E... rest)</code>           | Creates an enum set initially containing the specified elements                                                  |
| <code>static &lt;E extends Enum&lt;E&gt;&gt; EnumSet&lt;E&gt; of (E e1, E e2)</code>                   | Creates an enum set initially containing the specified elements                                                  |
| <code>static &lt;E extends Enum&lt;E&gt;&gt; EnumSet&lt;E&gt; of (E e1, E e2, E e3)</code>             | Creates an enum set initially containing the specified elements                                                  |
| <code>static &lt;E extends Enum&lt;E&gt;&gt; EnumSet&lt;E&gt; of (E e1, E e2, E e3, E e4)</code>       | Creates an enum set initially containing the specified elements                                                  |
| <code>static &lt;E extends Enum&lt;E&gt;&gt; EnumSet&lt;E&gt; of (E e1, E e2, E e3, E e4, E e5)</code> | Creates an enum set initially containing the specified elements                                                  |
| <code>static &lt;E extends Enum&lt;E&gt;&gt; EnumSet&lt;E&gt; range (E from, E to)</code>              | Creates an enum set initially containing all of the elements in the range defined by the two specified endpoints |

Z.B.:

```
EnumSet<Faehigkeit> faehigkeiten = EnumSet.allOf(Faehigkeit.class)
```

## Das Interface **List** (1)

- Erweiterung von **Collection**
- konkrete Implementierungen
  - **ArrayList**, **Vector**, **Stack**



# Das Interface `List` (2)

`void add(int index, E element)`

Inserts the specified element at the specified position in this list (optional operation).

`boolean addAll(int index, Collection<? extends E> c)`

Inserts all of the elements in the specified collection into this list at the specified position (optional operation).

`E get(int index)`

Returns the element at the specified position in this list.

`int indexOf(Object o)`

Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

`int lastIndexOf(Object o)`

Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

`ListIterator<E> listIterator()`

Returns a list iterator over the elements in this list (in proper sequence).

`ListIterator<E> listIterator(int index)`

Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.

`E remove(int index)`

Removes the element at the specified position in this list (optional operation).

`E set(int index, E element)`

Replaces the element at the specified position in this list with the specified element (optional operation).

`List<E> subList(int fromIndex, int toIndex)`

Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

## Das Interface `Queue` (1)

- Erweiterung von `Collection`
- konkrete Implementierungen
  - `LinkedList`, `PriorityQueue`
- Operationen mit Exceptions oder speziellem Returnwert:

|         | Throws exception       | Returns special value |
|---------|------------------------|-----------------------|
| Insert  | <code>add(e)</code>    | <code>offer(e)</code> |
| Remove  | <code>remove()</code>  | <code>poll()</code>   |
| Examine | <code>element()</code> | <code>peek()</code>   |

- Einfügen von null-Werten sollte vermieden werden, da sonst der spezielle return-Wert null (`peek`, `poll`) sinnlos ist

## Das Interface Queue (2)

**E** **element**()

Retrieves, but does not remove, the head of this queue.

**boolean** **offer**(**E** e)

Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.

**E** **peek**()

Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

**E** **poll**()

Retrieves and removes the head of this queue, or returns null if this queue is empty.

## Das Interface **Deque** (1)

- Erweiterung von **Queue**, so dass auf beide Enden zugegriffen werden kann.
- konkrete Implementierungen
  - **LinkedList**, **ArrayDeque**

## Das Interface Deque (2)

**void** **addFirst**(E e)

Inserts the specified element at the front of this deque if it is possible to do so immediately without violating capacity restrictions.

**void** **addLast**(E e)

Inserts the specified element at the end of this deque if it is possible to do so immediately without violating capacity restrictions.

**Iterator**<E> **descendingIterator**()

Returns an iterator over the elements in this deque in reverse sequential order.

**E** **getFirst**()

Retrieves, but does not remove, the first element of this deque.

**E** **getLast**()

Retrieves, but does not remove, the last element of this deque.

**boolean** **offerFirst**(E e)

Inserts the specified element at the front of this deque unless it would violate capacity restrictions.

**boolean** **offerLast**(E e)

Inserts the specified element at the end of this deque unless it would violate capacity restrictions.

**E** **peekFirst**()

Retrieves, but does not remove, the first element of this deque, or returns null if this deque is empty.

**E** **peekLast**()

Retrieves, but does not remove, the last element of this deque, or returns null if this deque is empty.

**E** **pollFirst**()

Retrieves and removes the first element of this deque, or returns null if this deque is empty.

**E** **pollLast**()

Retrieves and removes the last element of this deque, or returns null if this deque is empty.

## Das Interface Deque (3)

**E** **pop()**

Pops an element from the stack represented by this deque.

**void** **push(E e)**

Pushes an element onto the stack represented by this deque (in other words, at the head of this deque) if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an `IllegalStateException` if no space is currently available.

**E** **removeFirst()**

Retrieves and removes the first element of this deque.

**boolean** **removeFirstOccurrence(Object o)**

Removes the first occurrence of the specified element from this deque.

**E** **removeLast()**

Retrieves and removes the last element of this deque.

**boolean** **removeLastOccurrence(Object o)**

Removes the last occurrence of the specified element from this deque.

# Definition und Instanziierung einer Collection

## Definition der Variablen

Verwendung einer möglichst grundlegenden Schnittstelle erlaubt das einfache Ersetzen durch eine beliebige andere konkrete Implementierung, die die Mindestanforderungen erfüllt

```
Collection<Person> cp = new LinkedList<Person>();
```

## Erzeugen eines Collection-Objekts mit new

Es kann eine beliebige konkrete Klassenimplementierung verwendet werden, die die erforderliche Schnittstelle anbietet

## Befüllen von Collections

- Neben der Möglichkeit, mit `add` und `addAll` einzelne Objekte, bzw. alle Objekte einer anderen Collection einzufüllen, bieten alle konkreten Implementierungen des Collection Frameworks Konstruktoren an (diese können im Interface nicht vorgeschrieben werden), die schon beim Erzeugen einer Collection alle Elemente einer anderen Collection übernehmen.

```
new HashSet<Person>(cp)
```

- Mittels der Methode `asList` (statische Methode der Klasse `Arrays`) kann eine Liste aus einem Array erzeugt werden.

```
Person[] pa = new Person[10];
```

```
... new PriorityQueue pq(Arrays.asList(pa)); ...
```



# Iterieren über eine Collection

- Methode 1: for-each Schleife

```
for (Person p : cp) ...
```

Inhalt der Collection kann nicht verändert werden

- Methode 2: Iterator

```
for (Iterator<Person> i = cp.iterator(); i.hasNext(); ) ... i.next()  
...
```

Aktuelles Objekt kann aus der Collection mittels der `remove` Methode der Klasse `Iterator` entfernt werden.

- Methode 3: `ListIterator`

```
for (ListIterator<Person> i = cp.listIterator(); i.hasNext(); ) ... i.next()  
...
```

Bietet mehr Möglichkeiten, die Collection zu manipulieren (siehe nächste Folien). Wird nicht von allen Collection-Klassen unterstützt.

- Methode 4: Indiziert

```
for (int i = 0; i < cp.size(); ++i) ... cp.get(i) ...
```

Nur möglich, wenn die Collection indizierten Zugriff erlaubt. Dann oft ineffizient.

# Das Interface `Iterator`

boolean `hasNext()`

Returns true if the iteration has more elements.

E `next()`

Returns the next element in the iteration.

void `remove()`

Removes from the underlying collection the last element returned by this iterator (optional operation).

# Das Interface `ListIterator`

`void add(Object o)`

Inserts the specified element into the list (optional operation).

`boolean hasNext()`

Returns true if this list iterator has more elements when traversing the list in the forward direction.

`boolean hasPrevious()`

Returns true if this list iterator has more elements when traversing the list in the reverse direction.

`Object next()`

Returns the next element in the list.

`int nextIndex()`

Returns the index of the element that would be returned by a subsequent call to next.

`Object previous()`

Returns the previous element in the list.

`int previousIndex()`

Returns the index of the element that would be returned by a subsequent call to previous.

`void remove()`

Removes from the list the last element that was returned by next or previous (optional operation).

`void set(Object o)`

Replaces the last element returned by next or previous with the specified element (optional operation).

## Algorithmen für Collections

- In der Klasse **Collections** werden eine Vielzahl von immer wieder benötigten Funktionalitäten angeboten.
- Aufruf mittels **Collections.XXXXX**
- Die zu bearbeitende Collection wird als Parameter übergeben
- Hier nur eine Auswahl von besonders interessanten (bei einigen wird vorausgesetzt, dass es ein Ordnungskriterium für die Liste gibt, bzw. dass die Liste nach eben diesem Kriterium aufsteigend sortiert ist)

# Algorithmen für Collections (ausgewählte Methoden)

`static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)`

Searches the specified list for the specified object using the binary search algorithm.

`static <T> int binarySearch(List(List<? extends T> list, T key, Comparator<? super T> c)`

Searches the specified list for the specified object using the binary search algorithm.

`static <T> void copy(List<? super T> dest, List<? extends T> src)`

Copies all of the elements from one list into another.

`static boolean disjoint(Collection<?> c1, Collection<?> c2)`

Returns true if the two specified collections have no elements in common.

`static int frequency(Collection<?> c, Object o)`

Returns the number of elements in the specified collection equal to the specified object.

`static <T extends  
Object & Comparable<?  
super T>>  
T`

`max(Collection<? extends T> coll)`

Returns the maximum element of the given collection, according to the natural ordering of its elements.

`static <T extends  
Object & Comparable<?  
super T>>  
T`

`min(Collection<? extends T> coll)`

Returns the minimum element of the given collection, according to the natural ordering of its elements.

`static void reverse(List<?> list)`

Reverses the order of the elements in the specified list.

`static void rotate(List<?> list, int distance)`

Rotates the elements in the specified list by the specified distance.

`static void shuffle(List<?> list)`

Randomly permutes the specified list using a default source of randomness.

`static void shuffle(List<?> list, Random rnd)`

Randomly permute the specified list using the specified source of randomness.

# Maps

- Eine Map stellt eine Verallgemeinerung des Array-Konzepts dar, bei welchem der Index ein beliebiger Datentyp sein kann.
- Jedes gespeicherte Objekt kann über seinen Schlüssel schnell zugegriffen werden.
- Beispiele: Person/Name, Person/SVNr, deutsches Wort/englisches Wort (daher auch die alternative Bezeichnung dictionary).
- Eigentlich immer eine 1:1 Abbildung. Mittels Verwendung von Collections ist auch eine 1:n Abbildung möglich.
- Keine Collection im eigentlichen Sinn, da Wertepaare verwaltet werden. Wurde aber in Java, wegen der vielen Ähnlichkeiten in das Collection Framework aufgenommen.
- Implementierung oft mittels Hashtabellen.

# Das Interface **Map** (1)

- **konkrete Implementierungen**

- **HashMap**, (**HashTable** (synchronisiert))

`void clear()`

Removes all of the mappings from this map (optional operation).

`boolean containsKey(Object key)`

Returns true if this map contains a mapping for the specified key.

`boolean containsValue(Object value)`

Returns true if this map maps one or more keys to the specified value.

`Set<Map.Entry<K,V>> entrySet()`

Returns a Set view of the mappings contained in this map.

`boolean equals(Object o)`

Compares the specified object with this map for equality.

`V get(Object key)`

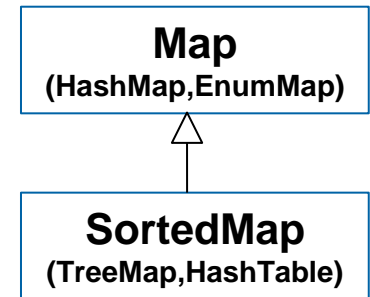
Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

`int hashCode()`

Returns the hash code value for this map.

`boolean isEmpty()`

Returns true if this map contains no key-value mappings.



## Das Interface **Map** (2)

**Set<K> keySet()**

Returns a Set view of the keys contained in this map.

**V put(K key, V value)**

Associates the specified value with the specified key in this map (optional operation).

**void putAll(Map<? extends K, ? extends V> m)**

Copies all of the mappings from the specified map to this map (optional operation).

**V remove(Object key)**

Removes the mapping for a key from this map if it is present (optional operation).

**int size()**

Returns the number of key-value mappings in this map.

**Collection<V> values()**

Returns a Collection view of the values contained in this map.



# Das Interface **SortedMap**

- **Map** bei der die **Keys** **sortiert** gespeichert werden
- **konkrete Implementierungen**
  - **TreeMap**, **HashTable** (synchronisiert)

`Comparator<? super K> comparator()`

Returns the comparator used to order the keys in this map, or null if this map uses the natural ordering of its keys.

`Set<Map.Entry<K,V>> entrySet()`

Returns a Set view of the mappings contained in this map.

`K firstKey()`

Returns the first (lowest) key currently in this map.

`SortedMap<K,V> headMap(K toKey)`

Returns a view of the portion of this map whose keys are strictly less than toKey.

`Set<K> keySet()`

Returns a Set view of the keys contained in this map.

`K lastKey()`

Returns the last (highest) key currently in this map.

`SortedMap<K,V> subMap(K fromKey, K toKey)`

Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.

`SortedMap<K,V> tailMap(K fromKey)`

Returns a view of the portion of this map whose keys are greater than or equal to fromKey.

`Collection<V> values()`

Returns a Collection view of the values contained in this map.

# Serialisierung

- Unter der Serialisierung von Objekten versteht man die Umwandlung des Objektzustandes (Inhalt von Instanzvariablen) in ein Bitmuster ("Serie von Bits"), aus dem der Objektzustand später wieder rekonstruiert werden kann.
- Wird z.B. verwendet, um Objekte in Dateien oder Datenbanken speichern zu können oder als Parameter in RMI-Aufrufen zu versenden

# Serialisierung und Marshalling

- Die beiden Begriffe werden in vielen Bereichen synonym verwendet
- Der Java Language Specification definiert aber Marshalling als eine Art der Serialisierung, bei welcher zusätzlich zum Objektinhalt auch die sogenannte code base, die den gesamten benötigten Code der Klasse (z.B. Methoden) enthält, mitcodiert wird.

# Serialisierung in Java

- Die JVM unterstützt Serialisierung weitgehend automatisch.
- Klassen müssen das Interface **Serializable** anbieten, um serialisierbar zu sein. Dies ist nur ein Marker-Interface und enthält keine Methodendeklarationen.
- In der Regel werden alle nicht statischen Attribute der Klasse serialisiert. Daher müssen alle Attribute selbst auch serialisierbar sein (primitive Datentypen oder Typen, die auch **Serializable** implementieren).
- Statische Attribute und solche, die als **transient** markiert sind, werden nicht serialisiert. Sie werden beim Lesen automatisch auf die entsprechenden Defaultwerte gesetzt.
- Für spezielle Anforderungen kann eine serialisierbare Klasse die Methoden **writeObject** und **readObject** definieren, um das Standardverhalten der Java Serialisierung zu überladen.

# Serialisierung und Vererbung

- Eine erbende Klasse kann auch dann serialisierbar sein, wenn die Elternklasse nicht serialisierbar ist.
- In diesem Fall muss die Elternklasse einen Defaultkonstruktor aufweisen, der zur Konstruktion des Teilobjekts der Elternklasse herangezogen wird.

# Versionierung

- Versionierung verhindert, dass bei Änderungen der Klasse ein älteres gespeichertes Objekt in die inkompatible neue Klassenversion geladen wird bzw. umgekehrt.
- Dazu definieren Klassen eine private, statische Konstante **serialVersionUID** vom Typ long, die verändert werden muss, wenn Änderungen der Klasse die neuen Objekte inkompatibel zu den alten Objekten machen.
- Diese Konstante kann von Eclipse automatisch generiert werden und es wird oft ein Hashcode der Klassendefinition selbst verwendet. Jeder beliebige Wert ist aber in Ordnung, soweit der Wert bei entsprechenden Klassenänderungen verändert wird.

# Kompatible Änderungen

- Erfordern keine Veränderung der Versionierungsnummer
- Attribute hinzufügen – werden mit Defaultwerten aufgefüllt
- Klassen hinzufügen – Felder der Klasse werden mit Defaultwerten aufgefüllt
- Klassen entfernen – Entsprechende Felder werden einfach ignoriert (Teilobjekte werden erzeugt, da sie eventuell in der weiteren Serialisierung referenziert werden. Sie werden durch den garbage collector wieder zerstört.)
- Hinzufügen von `writeObject/readObject` Methoden – Die Methoden werden wie erwartet aufgerufen
- Entfernen von `writeObject/readObject` Methoden – Die entsprechenden Daten werden einfach ignoriert
- Hinzufügen von `java.io.Serializable` – Entspricht dem Hinzufügen von Klassen. Felder werden mit Defaultwerten aufgefüllt
- Zugriffsrechte auf ein Feld ändern – Die access modifier `public`, `package`, `protected`, und `private` haben keinen Einfluss auf die Serialisierung der Felder
- Ändern eines Feldes von static auf nonstatic oder transient auf nontransient – Verhält sich wie beim Hinzufügen von Attributen

# Inkompatible Änderungen

- Löschen von Attributen – Wird ein Objekt der neuen Klassenversion in ein Objekt der alten Version eingelesen, wird das Attribut auf den Defaultwert gesetzt. Das verletzt eventuell die Integrität des Objekts.
- Klassen in der Hierarchie verschieben – Die Reihenfolge der serialisierten Daten würde nicht stimmen.
- Ändern eines Feldes von nonstatic auf static or von nontransient auf transient – Entspricht dem Löschen eines Attributs.
- Den Datentyp eines primitiven Felds ändern – Frühere Klassenversionen werden das Feld wegen des falschen Datentyps nicht lesen können.
- Ändern von `writeObject` oder `readObject` so, dass die Default-Daten (die von Java normalerweise automatisch verwaltet werden) anders behandelt werden, als zuvor.
- Ändern einer Klasse von `Serializable` zu `Externalizable` oder umgekehrt.
- Ändern einer Klasse von non-enum Typ zu einem enum Typ oder umgekehrt.
- Entfernen von `Serializable` oder `Externalizable`.
- Hinzufügen einer `writeReplace` oder `readResolve` Methode

spezielle  
Serialisierungsmethoden

Alternative Schnittstelle zur Serialisierung. Kontrolle bei Programmierern



## Beispiel: Objekt in Datei schreiben

```
import java.io.*;

public class SerializeDemo {
    public static void main(String [] args) {
        Employee e = new Employee();
        ... //e mit Daten befüllen
        try {
            FileOutputStream fileOut = new FileOutputStream("employee.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(e);
            out.close();
            fileOut.close();
        }
        catch(IOException i) { i.printStackTrace(); }
    }
}
```

# Beispiel: Objekt von Datei lesen

```
import java.io.*;

public class DeserializeDemo {
    public static void main(String [] args) {
        Employee e = null;
        try {
            FileInputStream fileIn = new FileInputStream("employee.ser");
            ObjectInputStream in = new ObjectInputStream(fileIn);
            e = (Employee) in.readObject();
            in.close();
            fileIn.close();
        } catch (IOException i) {
            i.printStackTrace(); return;
        } catch (ClassNotFoundException c) {
            System.out.println("Employee class not found");
            c.printStackTrace(); return;
        }
        ... //Employee Objekt verwenden
    }
}
```

Typumwandlung notwendig,  
da readObject immer ein  
Object liefert.



universität  
wien

# Anhang

# Iterator Invalidierung: Einfügen (Insertion) 1

| Container                  | Rules                                                                                                                                                                                                       | n3242 ref. |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|
| <b>Sequence containers</b> |                                                                                                                                                                                                             |            |
| vector                     | all iterators and references before the point of insertion are unaffected, unless the new container size is greater than the previous capacity (in which case all iterators and references are invalidated) | 23.3.6.5/1 |
| deque                      | all iterators and references are invalidated, unless the inserted member is at an end (front or back) of the deque (in which case all iterators are invalidated, but references to elements are unaffected) | 23.3.3.4/1 |
| list                       | all iterators and references unaffected                                                                                                                                                                     | 23.3.5.4/1 |
| forward_list               | all iterators and references unaffected ( <i>applies to insert_after</i> )                                                                                                                                  | 23.3.4.5/1 |
| array                      | (n/a)                                                                                                                                                                                                       |            |

# Iterator Invalidierung: Einfügen (Insertion) 2

| Associative containers           |                                                                            |          |
|----------------------------------|----------------------------------------------------------------------------|----------|
| set                              | all iterators and references unaffected                                    | 23.2.4/9 |
| multiset                         |                                                                            |          |
| map                              |                                                                            |          |
| multimap                         |                                                                            |          |
| Unordered associative containers |                                                                            |          |
| unordered_set                    | all iterators invalidated when rehashing occurs, but references unaffected | 23.2.5/8 |
| unordered_multiset               |                                                                            |          |
| unordered_map                    |                                                                            |          |
| unordered_multimap               |                                                                            |          |
| Container adaptors               |                                                                            |          |
| stack                            | inherited from underlying container                                        |          |
| queue                            |                                                                            |          |
| priority_queue                   |                                                                            |          |

# Iterator Invalidierung: Löschen (Erasure) 1

| Container                  | Rules                                                                                                                                                                                                                                                                                                                          | n3242 ref. |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|
| <b>Sequence containers</b> |                                                                                                                                                                                                                                                                                                                                |            |
| vector                     | every iterator and reference after the point of erase is invalidated                                                                                                                                                                                                                                                           | 23.3.6.5/3 |
| deque                      | erasing the last element invalidates only iterators and references to the erased elements and the past-the-end iterator; erasing the first element invalidates only iterators and references to the erased elements; erasing any other elements invalidates all iterators and references (including the past-the-end iterator) | 23.3.3.4/1 |
| list                       | only the iterators and references to the erased element is invalidated                                                                                                                                                                                                                                                         | 23.3.5.4/1 |
| forward_list               | only the iterators and references to the erased element is invalidated ( <i>applies to erase_after</i> )                                                                                                                                                                                                                       | 23.3.4.5/1 |
| array                      | (n/a)                                                                                                                                                                                                                                                                                                                          |            |

# Iterator Invalidierung: Löschen (Erasure) 2

|                                  |                                                                      |           |
|----------------------------------|----------------------------------------------------------------------|-----------|
| Associative containers           |                                                                      |           |
| set                              | only iterators and references to the erased elements are invalidated | 23.2.4/9  |
| multiset                         |                                                                      |           |
| map                              |                                                                      |           |
| multimap                         |                                                                      |           |
| Unordered associative containers |                                                                      |           |
| unordered_set                    | only iterators and references to the erased elements are invalidated | 23.2.5/13 |
| unordered_multiset               |                                                                      |           |
| unordered_map                    |                                                                      |           |
| unordered_multimap               |                                                                      |           |
| Container adaptors               |                                                                      |           |
| stack                            | inherited from underlying container                                  |           |
| queue                            |                                                                      |           |
| priority_queue                   |                                                                      |           |

# Iterator Invalidierung: Größenänderung (Resizing)

| Container                  | Rules               | n3242 ref.  |
|----------------------------|---------------------|-------------|
| <b>Sequence containers</b> |                     |             |
| vector                     | as per insert/erase | 23.3.6.5/12 |
| deque                      | as per insert/erase | 23.3.3.4/1  |
| list                       | as per insert/erase | 23.3.5.3/1  |
| forward_list               | as per insert/erase | 23.3.4.5/25 |
| array                      | (n/a)               |             |



# Iterator Invalidierung: Bemerkungen

## Footnote 1

Unless otherwise specified (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to a library function shall not invalidate iterators to, or change the values of, objects within that container. [23.2.1/11]

## Footnote 2

No swap() function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. [ Note: The end() iterator does not refer to any element, so it may be invalidated. —end note ] [23.2.1/10]

## Footnote 3

Other than the above caveat regarding swap() and the erasure rule for deque, it's not clear whether "end" iterators are subject to the above listed per-container rules; you should assume, anyway, that they are.

# Lambdaausdrücke (lambda expressions)

- Ein Ausdruck, der eine Funktion definiert.

```
auto i = find_if(v.begin(), v.end(), [] (int val) {...});
```

- +Definition an der Stelle der Verwendung
- +Syntax einfacher als bei Funktionspointern oder Funktoren (Funktionsobjekten)
- -Temporäres Objekt, daher kein dauerhafter Zustand

# Lambdaexpressions Syntax (1)

- Seit C++11 eine neue syntaktische Möglichkeit, bequem Funktionsobjekte zu verwenden:

```
[ ] ( ) { capture Ausdruck (siehe später)
      Parameterliste
      Funktionsblock }
```

- Der Typ des Returnwerts (im konkreten Fall void) kann vom Compiler automatisch ermittelt werden, soweit nicht mehrere Returnstatements im Block vorkommen (manche Compiler finden auch bei mehreren Returnstatements einen passenden Returntyp, aber der Standard garantiert das nicht).
- Angabe des Returntyps ist auch explizit möglich:

```
[ ] ( ) -> void { Returntyp cout << "Hello world"; }
```

## Lambdaexpressions Syntax (2)

- Auch eine „exception specification“ kann angegeben werden. In C++11 gelten allerdings „exception specifications“ als überholt (deprecated).

```
[] () throws() {cout << "Hello world";}
```

exception specification

- Auch noexcept ist möglich:

```
[] () noexcept {cout << "Hello world";}
```

Funktion darf keine Exception werfen

# Lambdaexpressions Closures

- Ein Lambda-Ausdruck kann Variablen, die in der umgebenden Funktion zugreifbar (in scope) sind, referenzieren (in einer Methode damit auch Instanzvariablen der entsprechenden Klasse)

```
class X {  
    int x;  
public:  
    void methode() {  
        int n;  
        [=] {cout << n << x;}; //this->x  
    }  
};
```

Damit das funktioniert, muss das erzeugte Funktionsobjekt auch die gesamte (referenzierte) Umgebung beinhalten. Im Allgemeinen wird das erreicht, indem die entsprechenden Variablen vom Stack auf den Heap transferiert werden und dafür Sorge getragen wird, dass der Zugriff auf die Variablen von verschiedenen Seiten weiterhin funktioniert. Das Funktionsobjekt schließt somit die umgebenden Variablen mit ein. Daher der Begriff **Closure**.

# Lambdaexpressions Capture Klausel

- Die „capture clause“ legt fest, wie die vom Lambda Ausdruck verwendeten Variablen im Funktionsobjekt gespeichert werden:

```
[ ]           //kein capture - kein Zugriff auf Variablen der Umgebung

[var1, &var2, ...] //explizite Liste der zugreifbaren Variablen
                  //& bedeutet, dass Variable als Referenz übernommen wird
                  //nicht mehr verwendbar, wenn Variablen out of scope sind!
                  //andere Variable werden in konstante Instanzvariable des
                  //Funktionsobjekts kopiert (mutable notwendig, um diese
                  //ändern zu können)

[=, &var1, ...]  //Defaultübernahme per Kopie, die explizit angegebenen
                  //Variablen müssen alle Referenzen sein

[&, var1, ...]   //Defaultübernahme als Referenz, die explizit angegebenen
                  //Variablen müssen alle Kopien sein

[this]          //Schlüsselwort this kann verwendet werden, um explizit
                  //Zugriff auf Instanzvariablen zu erhalten (nur konstant
                  //möglich; Instanzvariablen werden per Pointer zugegriffen)
                  //this ist automatisch in = oder & inkludiert
```

# Lambdaexpressions Typ

- Der Typ eines Lambda-Ausdrucks hat keine syntaktische Entsprechung. Um eine Variable, einen Parameter oder einen Returnwert vom passenden Typ zu definieren, wird daher `auto`, `std::function` oder ein Template verwendet:

```
#include<functional> //definiert std::function

std::function<int (int)> g(std::function<void ()> f) {
    f();
    return [] (int n) {return n*2;};
}

int main() {
    auto func = [] () -> void {cout << "Hello world\n";};
    std::function<void ()> func2;
    func2 = func;
    auto h = g(func2);
    cout << h(3) << '\n';
}
```

`std::function` ist verträglich mit Lambdaausdrücken, Funktoren und Funktionspointern!



# Templates ein simples Beispiel (1)

Templates bilden die Möglichkeit, die Typen von Variablen und Funktionsparametern zu parametrisieren (parametrischer Polymorphismus):

```
template <typename E>  
E max(E a, E b) {  
    return a > b ? a : b;  
}
```

**E** ist ein Typparameter und wird bei Bedarf durch einen konkreten Typ ersetzt.  
(Es können beliebig viele Typparameter mit Komma getrennt angegeben werden)

Dieses Template kann für alle Datentypen verwendet werden, die einen passenden Operator `>` unterstützen.

Verwendung:

```
max(7, 5);  
max(1.7, 9.1);  
max<int>(7, 5);  
max<long>(3, 10.5);  
max<double>(3, 10.5);  
...  
max<const char*>("abc", "xyz");
```

Problem!



## Templates ein simples Beispiel (2)

Für bestimmte Werte (Typen) von Typparametern können spezielle Versionen der Templatefunktion definiert werden:

```
template <>
const char *max(const char *a, const char *b)
{
    return strcmp(a,b)>0 ? a : b;
}
```

Hier werden nur die Typparameter  
angeführt, die in dieser **Spezialisierung**  
nicht durch konkrete Typen ersetzt wurden

Verwendung:

```
max("abc", "xyz");
max<const char*>("abc", "xyz");
```

# Template Klassen

- Wird eine Template Klasse verwendet, so sind alle Methoden der Klasse implizit Templatefunktionen

```
template<typename T>
class Vector {
    size_t max_sz;
    size_t sz;
    T* element;
public:
    void push_back(const T&);
    ...
};

template<typename T>
Vector<T>::push_back(const T& val) {...}
```

# Container speichern Kopien (1)

```
#include<iostream>
#include<vector>
#include<unordered_map>
#include<stdexcept>
using namespace std;

class Person {
    string vorname;
    string zuname;
    int alter;
public:
    Person(string vorname, string zuname, int alter) : vorname{vorname}, zuname{zuname}, alter{alter} {}
    string getName() const {return zuname+" "+vorname;}

    int getAlter() const {return alter;}
    void setAlter(int alter) {
        if (alter<0) throw runtime_error("alter muss >= 0 sein!");
        this->alter = alter;
    }
    ostream& print(ostream& o) const {
        return o<<'['<<zuname<<' '<<vorname<<' '<<alter<<']';
    }
};

ostream& operator<<(ostream&o, const Person& p) {
    return p.print(o);
}
```

## Container speichern Kopien (2)

```
int main() {  
    vector<Person> v {Person{"Hans","Bauer",36}, Person{"Lisa","Mueller", 25}, Person{"Maria", "Mayer",40}};  
    unordered_map<string, Person> m;  
    for (const auto& p : v) m.insert({p.getName(),p}); //std::make_pair(p.getName(),p)  
    v[1].setAlter(22);  
    for (const auto& p : v) cout << p << " ";  
    cout << '\n';  
    for (const auto& pair : m) cout << pair.second << " ";  
    cout << '\n';  
  
    return 0;  
}
```

Ausgabe:

```
[Bauer Hans 36] [Mueller Lisa 22] [Mayer Maria 40]  
[Mayer Maria 40] [Mueller Lisa 25] [Bauer Hans 36]
```

## Container speichern Kopien (3)

```
#include<iostream>
#include<vector>
#include<unordered_map>
#include<stdexcept>
using namespace std;

class Person {
    string vorname;
    string zuname;
    int alter;
public:
    Person(string vorname, string zuname, int alter) : vorname{vorname}, zuname{zuname}, alter{alter} {}
    virtual ~Person() {}
    string getName() const {return zuname+" "+vorname;}
    int getAlter() const {return alter;}
    void setAlter(int alter) {
        if (alter<0) throw runtime_error("alter muss >= 0 sein!");
        this->alter = alter;
    }
    virtual ostream& print(ostream& o) const {
        return o<< '['<<zuname<< ' '<<vorname<< ' '<<alter<<']';
    }
};
```

## Container speichern Kopien (4)

```
class Angestellte: public Person {
    int personal_nummer;
public:
    Angestellte(string vorname, string zuname, int alter, int personal_nummer) : Person{vorname, zuname, alter},
        personal_nummer{personal_nummer} {}

    ostream& print(ostream& o) const override {
        o<<'* '<<personal_nummer<<'* ' ;
        return Person::print(o);
    }
};

ostream& operator<<(ostream&o, const Person& p) {
    return p.print(o);
}
```

## Container speichern Kopien (5)

```
int main() {  
    vector<Angestellte> va {Angestellte{"Hans","Bauer",36,1}, Angestellte{"Lisa","Mueller",25,2},  
                           Angestellte{"Maria", "Mayer",40,3}};  
    vector<Person> vp {va.begin(), va.end()};  
  
    for (const auto& p : va) cout << p << " ";  
    cout << '\n';  
    for (const auto& p : vp) cout << p << " ";  
    cout << '\n';  
  
    return 0;  
}
```

Ausgabe:

```
*1*[Bauer Hans 36] *2*[Mueller Lisa 25] *3*[Mayer Maria 40]  
[Bauer Hans 36] [Mueller Lisa 25] [Mayer Maria 40]
```

# Java Annotations (1)

- Möglichkeit, in Java Programm-Metainformationen (Informationen über das Programm) zu notieren.
- Java definiert 10 Annotationen vor.
  - eigentliche Annotationen
    - `@Override`
    - `@Deprecated`
    - `@SuppressWarnings(value={<werteliste>})`
      - erlaubte Werte: "deprecation", "unchecked", "path", "finally", ... (compilerabhängig)
    - `@SafeVarargs` (seit Java 7)
    - `@FunctionalInterface` (seit Java 8)
  - Annotationen für Annotationen (legen Eigenschaften von Annotationen fest, z.B. wenn man eigene definiert)
    - `@Documented`
    - `@Inherited`
    - `@Retention` (Werte: SOURCE, CLASS, RUNTIME)
    - `@Target` (Werte: ANNOTATION\_TYPE, CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE)
    - `@Repeatable` (seit Java 8)



## Java Annotations (2)

- Annotationen können bei Bedarf selbst definiert werden. Unterschiedliche Tools (Editor, Frameworks, Dokumentationsgenerator etc.) verwenden eine große Anzahl von Annotationen.

```
/**  
 * @param args  
 */  
public static void main(String[] args) {
```

Beschreibung eines Parameters einer Methode.  
Wird zum Generieren von Dokumentation durch Javadoc verwendet.

# Kovarianz und Kontravarianz

- Kovarianz und Kontravarianz beschreiben die Variation von Return- und Parametertypen beim Override von Methoden in Bezug auf die Vererbungshierarchie. Kovariant heißt, dass der verwendete Datentyp in der erbenden Klasse vom entsprechenden Datentyp in der Elternklasse (direkt oder indirekt) erben muss. Kontravariant bedeutet, dass der verwendete Datentyp in der erbenden Klasse eine (direkte oder indirekte) Basisklasse des entsprechenden Typs in der Elternklasse sein muss.
- Allgemein gilt: Um das LSP zu erfüllen, müssen beim Override von Methoden die Returntypen kovariant und die Parametertypen kontravariant sein. ("demand no more, provide no less")
- Sowohl in Java als auch in C++ werden aber nur kovariante Returntypen unterstützt. Kontravariante Parametertypen sind nicht möglich und führen zum Überladen (Overload) statt Übersteuern (Override)