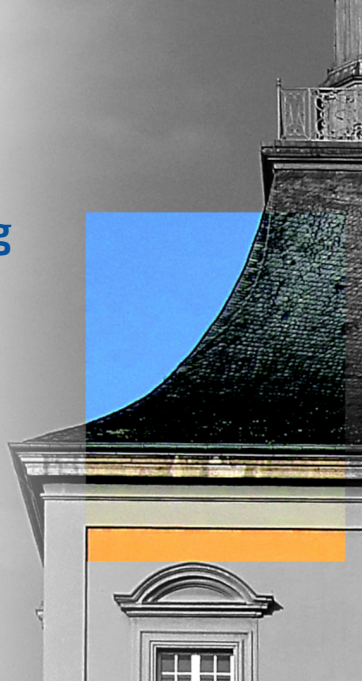
UNIVERSITÄT BONN

Algorithmen und Programmierung

Objektorientierte Programmierung

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik

Gitversion: '1dcd24136270d23b7189dc6922461c79c876ddb0'



Generische Programmierung

Erinnerung

Container und Sortierverfahren benötigen zwar einen Typ (zum Speichern / Sortieren), die Speicher- oder Sortierkonzepte hängen aber nicht von der genauen Ausprägung des Typs ab

Offne Frage

Wie drückt man aus, dass ein Konzept von einer festen Anzahl von Typen abhängt, die Ausprägung der Typen aber keine Rolle spielt?

UNIVERSITÄT BONN

Motivation

Funktionen fassen Programmcode zusammen, der unabhängig von expliziten Parameterwerten Sinn macht.

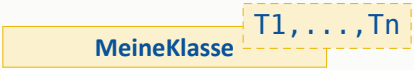
Mit Containertypen haben wir Konzepte gesehen, die von einer feste Anzahl von Typen abhängen, aber unabhängig von den expliziten Typen sind. Diese „variablen“ Typen wollen wir **Typparameter** nennen.

Wir wünschen uns also Programme zu schreiben, die Typparametern zulassen und somit unabhängig von explizit gegebenen Typen Sinn macht. Solche Programme heißt **generisch**.

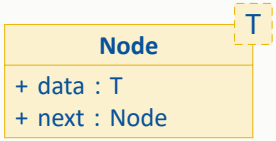
UNIVERSITÄT BONN

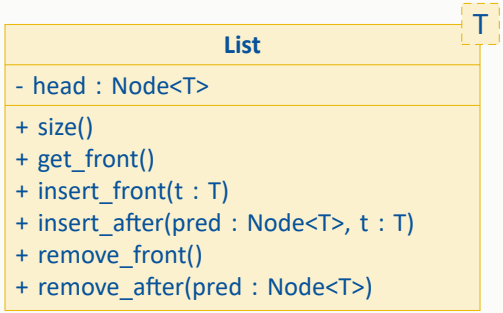
Generische Programmierung in UML

Um auszudrücken, dass eine Klasse von den Typparametern T1, . . ., Tn abhängt, werden diese Typparameter wie folgt notiert.



Beispielsweise werden Listen durch diese generischen Datenstruktur realisiert.





UNIVERSITÄT BONN

Generische Programmierung in objektorientierten Sprachen

Stark typisierte, objektorientierte Programmiersprachen unterscheiden sich kaum beim Entwurf von grundlegenden, generischen Programmen. Allerdings gibt es bei der Umsetzung von generischen Programmierkonzepten große Unterschiede. Diese führen zu sehr unterschiedlichen, fortgeschrittenen generischen Konzepten.

- **Java** stellt mit **Generics** die Möglichkeit für generische Programmierung zur Verfügung. Vereinfacht gesagt wird der generische Code einmal für den übergeordneten Typ `Object` kompiliert und zur Laufzeit durch Typecastings für alle Typen verfügbar.
- **C++** stellt mit **Templates** die Möglichkeit für generische Programmierung zur Verfügung. Hier wird der generische Code einmal pro verwendetem Typparameter kompiliert. Zusammen mit **Template Spezialisierung** wird mehr Flexibilität und Laufzeitperformance erreicht, aber man benötigt längerer Compilezeit und erhält ggf. größeren Programmdateien oder Bibliotheken.
- **Python** ist nicht stark typisiert. Einfach gesagt ist alles generisch.

UNIVERSITÄT BONN

Erste Klassentemplate definieren

Um in C++ eine generische Klasse X mit einem Typparameter T zu beschreiben, geht man wie folgt vor.

```
template<class T> // Innerhalb der generischen Klasse X ist T ein bekannter Typ
class X {
private:
  T data;
};
```

Um einen neuen Typ aus dem Template zu generieren, gibt man den Klassennamen inklusive Typparameter an.

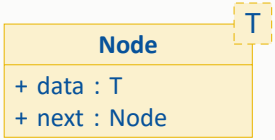
```
X<int>    x1; Typ: X mit Typparameter int
X<double> x2; Typ: X mit Typparameter double
X<X<int>> x3; Typ: X mit Typparameter X<int>
```

Es können auch mehrere Typparameter gefordert werden.

UNIVERSITÄT BONN

Beispiel

Wir sind nun in der Lage verkettete Listen mit Typparametern in C++ zu definieren.




```
template<class T>
struct Node {
  T data;
  std::shared_ptr<Node<T>> next;
};

template<class T>
class List {
public:
  typedef std::shared_ptr<Node<T>> Nodeptr;

  List();
  Nodeptr insert_front(const T& x);
  Nodeptr insert_after(const Nodeptr& pred, const T& x);
  /* Restliche Memberfunktionen */
private:
  Nodeptr head;
};
```

UNIVERSITÄT BONN

Templates der C++-Standardbibliothek

Der C++ Standard stellt unter anderem Templates für Container, Iteratorenund Algorithmen zur Verfügung. Zum Beispiel:

```
// Container
template<class T,   /* ... */>          class std::vector;
template<class Key, /* ... */>          class std::set;
template<class Key, class T, /* ... */> class std::map;
template<class Key, /* ... */>          class std::unordered_set;
template<class Key, class T, /* ... */> class std::unordered_map;
/* ... */

// Iteratoren
template</* ... */> struct std::iterator;

// Algorithmen
template<class RandomIt, class Compare>
void std::sort(RandomIt first, RandomIt last, Compare comp);
```

UNIVERSITÄT BONN

Umsetzung unterhalb von C++

Jedes Klassentemplate ist eine Schablone für eine (unendliche) Familie von Typen. Durch die Festlegung der Typparameter wird ein neuer Typ zur Verwendung vorgesehen. Man sagt, dass das Klassentemplate **instanziiert**[1] wird. Da jede Templateinstanz einem neuen Typ entspricht, nennt man Templates auch manchmel „Type Generators“.

Beim Instanziieren eines Templates geht der Compiler wie folgt vor. Für jede Templateinstanz wird eine Kopie des Templatecodes angelegt in der anschließend die expliziten Typparameter ersetzt werden. Anschließend wird jede Templateinstanz kompiliert.

[1] Der neue Typ ist also eine Instanz des Klassentemplates. Rein sprachlich sollen Programmieranfänger:innen instanziierte Templates (also neue Typen) nicht mit instanziierten Objekten (also Instanzen einer Klasse) verwechseln.

UNIVERSITÄT BONN

Notwendige Codeorganization

Da der Compiler bei jeder Templateinstanz einen neuen Typ anlegt und dazu die gesamten Implementierungsdetails des Templates benötigt, werden die Implementierungsdetails oft wie folgt abgelegt.

- Entweder werden die Implementierungsdetails des Templates im unteren Ende der Headerdatei zur Verfügung gestellt,
- oder die Implementierungsdetails werden in einer `.ipp`-Datei im `include`-Verzeichnis abgelegt und die Headerdatei endet mit dem Include dieser `.ipp`-Datei.

Würde man die Implementierungsdetails nur in einer separaten, nicht eingebundenen `.cpp`-Datei ablegen, dann wüsste der Compiler nicht notwendigerweise wie groß die Funktionsparameter oder lokalen Variablen der Memberfunktionen dieses Typs sind.

Zusammenfassung

Mit generischer Programmierung können Programme geschrieben werden, die nur von der Anzahl der Typparameter abhängen aber nicht von deren Ausprägung

In C++ wird generische Programmierung durch Templates ermöglicht. Hier wird zu jeder verwendeten Typparameterkombination ein neuer Typ angelegt und kompiliert

Haben Sie Fragen?

Generische Programmierung

Weitere grundlegende Konzepte

Ziel

Wir lernen Funktionstemplates, Templatespezialisierung und weitere grundlegende Konzepte kennen

UNIVERSITÄT BONN

Funktionstemplates

Neben Klassentemplates können auch **Funktionstemplates** definiert werden. Dies geschieht analog zur Definition der Klassentemplates.

```
template<class T>
void drucke_iterierbares(const T& x) {
  for(const auto& elem : x) {
    std::cout << elem << std::endl;
  }
}
```

Wenn die Templateparameter beim Funktionsaufruf durch die übergebenen Parameter automatisch abgeleitet werden können, dann dürfen die Templateparameter unterdrückt werden.

```
std::vector<int> v = ... // v ist vom Typ std::vector<int>
drucke_iterierbares(v);  // also drucke_iterierbares<std::vector<int>> verwendet.
```

Im Folgenden diskutieren wir **Templates** und meinen damit Klassentemplates oder Funktionstemplates.

UNIVERSITÄT BONN

Allgemeinere Template Parameter

Da Templatesinstanzen technisch als Kopie des Klassentemplates hervorgehen, ist es neben Typparametern auch möglich konstante Werte eines festen Typs als Templateparameter zu übergeben. In diesem Fall wird für jede Kombination von Typparametern und Wertparametern eine eigene Templateinstanz (also ein neuer Typ) angelegt.

Der Typ `std::array` ist ein Beispiel für ein Klassentemplate mit Wertparametern.

```
template<class T, size_t N>
struct array {
  ...
}
```

Für jede Kombination von Typparameter `T` und natürlicher Zahl `N` wird ein neuer Typ `std::array<T,N>` angelegt. Deshalb sind die Typen `std::array<double,3>` und `std::array<double,4>` unterschiedlich.

UNIVERSITÄT BONN

Abhängige Namen

Damit der Compiler auch vor der Templateinstanziierung prüfen kann, ob der Templatecode (möglichst) korrekt ist, möchte der Compiler so viele Informationen zu den beteiligten Typen haben wie möglich. Dazu verarbeitet der Compiler Namen. Ein Name ist **abhängig**, wenn er direkt oder indirekt von einem oder mehreren Templateparametern abhängt. Beispiel:

```
template<class T>
void f() {
  T::cooler_typ x; // T::cooler_typ ist ein abhänger Name
}
```

Das ein abhängiger Namen einen Typ benennen sollen, muss dem Compiler mitgeteilt werden. Dazu stellt man das Keyword **`typename`** voran.

```
template<class T>
void f() {
  typename T::cooler_typ x;
}
```

UNIVERSITÄT BONN

Defaultparameter für Templates

Es kommt vor, dass einige Templateparameter einen naheliegenden Default bereitstellen. Defaultparameter werden in der Templatedeklaration festgelegt. Der Defaultparameter muss in der Templateinstanziierung nicht angegeben werden, kann aber durch die Angabe eines anderen Parameters ersetzt werden.

```
template<class S, class T = /* Fester Typ */> ...
```

Ein Beispiel ist `std::unordered_set`.

```
template<
  class Key,                          // muss angegeben werden
  class Hash = std::hash<Key>,        // für die meisten Standardtypen definiert
  class KeyEqual = std::equal_to<Key>,  // verwendet operator== für den Vergleich
  class Allocator = std::allocator<Key> // kümmert sich um Speicherdinge
> class unordered_set;
```

Also kann man mit `std::unordered_set<int>` eine hashende Menge anlegen , ohne alle Templateparameter explizit anzugeben.

UNIVERSITÄT BONN

Templatespezialisierung I

Es kommt vor, dass ein Template für sehr wenige Typparameter oder Wertparameter unterschiedlich implementiert werden soll. Zum Beispiel aus Effizienzgründen. Dies ist durch **Templatespezialisierung** möglich. Hierzu gibt man die **partielle oder vollständige Templatedefinition** der Templateinstanz an.

Beispielsweise besitzt der Typ `template<class T> std::vector<class T>` die Templatespezialisierung `std::vector<bool>`. Diese verwendet pro gespeichertem `bool` nicht ein ganzes Byte, sondern fasst mehrere `bool`-Werte zu Byteblöcken zusammen.

UNIVERSITÄT BONN

Templatespezialisierung II

Im folgenden Beispiel wird ein Klassentemplate `A<class T,int I>` definiert. Dieses wird zu `A<class T, 2>` partiell spezialisiert. Anschließend wird dort eine Memberfunktionen spezialisiert. Bei der Templateinstanziierung wird immer das speziellste Template verwendet.

```
template<class T, int I>  // Klassentemplate
struct A {
  void f();               // Dekl von A<T,I>::f
};

template<class T, int I>
void A<T, I>::f() {}      // Defn von A<T,I>::f

template<class T> // Partielle Spezialisierung
struct A<T, 2> {  // von A<T,I> durch A<T,2>
  void f();
  void g();
  void h();
};
```

```
template<class T>
void A<T, 2>::g() {}     // Defn von A<T,2>::g

template<>               // Spezialisierung
void A<char, 2>::h() {}  // durch A<char, 2>::h

int main() {
  A<char, 0> a0; // Allgemein:  A<T=char, I=0>
  A<char, 2> a2; // Part. Spez: A<T=char, 2>
  a0.f(); // Allgemein:  A<T,I>::f
  a2.g(); // Part. Spez: A<T,2>::g
  a2.h(); // Part. Spez: A<char,2>::h
  a2.f(); // ERROR: A<T,2>::f nicht definiert
}
```

UNIVERSITÄT BONN

Explizite Templateinstanziierung I

Üblicherweise werden in jeder Compilationunit alle verwendete Templateinstanz vom Compiler übersetzt. Falls dieselbe Templateinstanz in verschiedenen Compilationunits verwendet wird, wird zusätzliche, unnötige Compilezeit verwendet.

Um diese Compilezeit zu verringern, können Templates in genau einer Compilationunit explizit instanziiert werden. Das ist insbesondere bei Bibliotheken mit „Standardtypen“ sinnvoll.

Beispielsweise ist der Typ `std::string` das explizite instanziierte Klassentemplate `std::basic_string<char>`.

Explizite Templateinstanziierung II

Zur expliziten Templateinstanziierung gehören zwei Schritte.

- In genau einer `.cpp`-Datei gibt man alle Templateinstanzen an, die man genau einmal Übersetzen möchte.
- In der Headerdatei des Templates gibt man an, welche Templateinstanzen nicht übersetzt werden sollen.

```
// Headerdatei
template<class T, ...> class X { ... };
...
// Der Compiler übersetzt dieses Template nicht in jeder Compilationunit
extern template class X<int,...>;
extern template class X<double,...>;


// Quelldatei
// Der Compiler übersetzt dieses Template in dieser Compilationunit
template class X<int,...>;
template class X<double,...>;
```

Zusammenfassung

Wir haben Funktionstemplates, Templatespezialisierung und weitere grundlegende Konzepte kennengelernt

Haben Sie Fragen?

Generische Programmierung

Generische Programmierung und Subtypenbeziehung

Offene Frage

Wie verhält sich generische Programmierung mit Subtypenbeziehungen?

UNIVERSITÄT BONN

Templates und Subklassen

Aus Templates werden neue Typen durch Templateinstanziierung. Wie bei üblichen Klassen können Subklassenbeziehnungen auch bei Klassentemplates ausgedrückt werden. Da Templateinstanziierung durch das Kopieren des Templates entsteht, dürfen sich die Templateparameter von Ober- und Unterklasse unterscheiden.

```
template<class T>
class B { /* ... */ };

template<class T>
class D1 : public B<T> { /* ... */ };

template<class S, class T>
class D2 : public B<S> { /* ... */ };

template<class T>
class D3 : public B<char> { /* ... */ };

class D4 : public B<int> { /* ... */ };
```

```
int main() {
  A<D> a;
  D1<int> d1;
  D2<int, char> d2;
  D3<int> d3;
  D4 d4;
}
```

UNIVERSITÄT BONN

Subtypenbeziehung zwischen Templateparametern

Man darf sich folgende Frage stellen. Wenn `X` ⊲— `Y` gegeben sind, und `A<T>` ein Klassentemplate ist , stehen dann `A<X>` und `A<Y>` immer in einer Subtypenbeziehung?

Kurz gesagt ist die Antwort **Nein**. Falls wir eine Funktion der Form `X A<X>::f(X)` haben, kann diese nicht von `Y A<Y>::f(Y)` überschrieben werden, denn Funktionsparameter verhalten sich kontravariant. Analog kann eine Funktion `Y A<Y>::f(Y)` nicht von `X A<X>::f(X)` überschrieben werden, denn Rückgabetypen verhalten sich kovariant.

Als Beispiel können sich merken, dass eine Liste von Katzen mehr fordert als eine Liste vom Tieren und eine Liste von Tieren weniger anbietet als eine Liste von Katzen.

Zusammenfassung

Typparameter, die in einer Subtypen Beziehnung stehen, führen NICHT zu einer Subtypen der instanziierten, generischen Typen

Anderenfalls wäre das Liskovsche Substitutionsprinzip verletzt

Haben Sie Fragen?

Iteratoren

Offne Frage

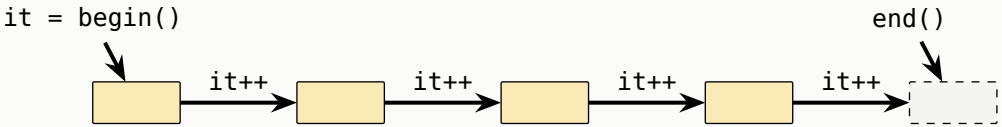
Wie iteriert man methodisch sinnvoll durch Container (und andere, iterierbare Objekte)?

UNIVERSITÄT BONN

Wiederholung

Ein **Iterator** referenziert entweder ein (zum Erstellungszeitpunkt des Iterators) enthaltenes Element Containers oder der Iterator enthält die Information, dass er auf kein Element des Containers zeigt. Der Iterator kann mit einer Laufzeit von $\mathcal{O}(\log(n))$ auf das nächste Element referenzieren oder, mit einer Laufzeit von $\mathcal{O}(1)$ die Information bereit stellen, dass alle Elemente einmal durchlaufen wurden.

Die Elemente eines allgemeinem Containers haben keine natürliche Reihenfolge. Ein allgemeiner Iterator durchläuft die Elemente eines allgemeinen Container in irgendeiner Reihenfolge.



UNIVERSITÄT BONN

Iteratoren in objektorientierte Sprachen

Iteratoren verhalten sich in objektorientierten Programmiersprachen sehr ähnlich, die jeweiligen Implementierungsdetails unterscheiden sich aber.

- In **Java** ist ein Objekt ein Iterator, falls es das generische Interface `Iterator<T>` implementiert.
- In **Python** ist ein Objekt ein Iterator, wenn es die Memberfunktionen `__next__` korrekt implementiert.
- In **C++** gibt es verschiedene Iteratoren. Aus unterschiedlichen Gründen[2] werden Iteratoren (auf Grund unseres aktuellen Wissenstands) ungewöhnlich implementiert, gleich mehr.

[2] Beispielsweise um bei generischen Algorithmen, die in `<algorithm>` implementiert sind, möglichst hohe Performance zu erzielen oder weil sich Iteratoren durch die in C++20 eingeführten `concepts` nachvollziehbarer beschreiben lassen.

UNIVERSITÄT BONN

Anforderungen an den iterierbaren Container in C++

Zum Durchlaufen der (konstanten) Elemente eines iterierbaren Container `C`, stellt dieser einen Iterator `it` mit `C.begin()` bzw. `C.cbegin()` bereit. Der Iterator `it` durchläuft die Elemente mithilfe der Operation `it++`. Der Iterator zeigt an, dass er das Ende erreicht hat, wenn er gleich dem Iterator `C.end` bzw. `C.cend()` ist.

UNIVERSITÄT BONN

Iteratorarten in C++

In C++ spricht man über die folgenden Iteratorarten und Iteratoroperationen.

Category	output	input	forward	bidirectional	random-access
Read		=*p	=*p	=*p	=*p
Access		->	->	->	-> []
Write	*p=		*p=	*p=	*p=
Iterate	++	++	++	++ --	++ -- + - += -=
Compare		== !=	== !=	== !=	== != < > >= <=

UNIVERSITÄT BONN

Anforderungen an Iteratoren I

Iteratoren werden bis C++17 als Subklasse dieser Klassentemplates implementiert[3].

```
template<
  class Category,       // Gibt an welche Art von Iterator implementiert wird
  class T,              // Grundlegender Elementtyp der durchlaufen wird
  class Distance = std::ptrdiff_t, // Nur für Arrayiteratoren relevant
  class Pointer = T*,   // Dereferenzierbares Objekt um auf Elemente zuzugreifen
  class Reference = T&  // Referenztyp zum Elementzugriff
> struct iterator;
```

Die übergebenen Templateparameter stehen uns durch die folgenden Membertypen zur Verfügung. `iterator_category`, `value_type`, `difference_type`, `pointer` und `reference`.

Die `Category` wird von generischen Algorithmen aus `<algorithm>` ausgelesen, um möglichst effiziente Implementierungsdetails auszuwählen. Die verfügbaren Iteratorarten findet man unter cppreference.com

[3] Ab C++20 verwendet man concepts um Iteratoren zu implementieren.

UNIVERSITÄT BONN

Anforderungen an Iteratoren II

Ein Forwarditerator muss mindestens die folgenden Member bereitstellen.

```
class mein_iterator : public std::iterator<
    std::forward_iterator_tag,  // iterator_category
    ElementTyp,                 // value_type
    std::ptrdiff_t,             // difference_type
    ElementTypPointer,          // pointer
    ElementTyp&                 // reference
  >
{
public:
  mein_iterator(/* Zur Initialisierung durch den Container */);
  mein_iterator& operator++();   // ++it
  mein_iterator operator++(int); // it++ [int ist ein Dummyparameter]
  bool operator==(mein_iterator other); // it == anderer_it
  bool operator!=(mein_iterator other); // it != anderer_it
  reference operator*();         // *it
};
```

UNIVERSITÄT BONN

Beispiel

Zur Demonstration implementieren wir einen Forwarditerator für unsere verkettete Liste. Hier ist der Elementtyp `Node` und der Pointertyp `std::shared_ptr<Node>`.

```
class mein_iterator : public std::iterator<
    std::forward_iterator_tag, // iterator_category
    Node,                      // value_type
    std::ptrdiff_t,            // difference_type
    Nodeptr,                   // pointer
    Node&                      // reference
  >
{
private:
  pointer _p;
public:
  mein_iterator(mein_iterator::pointer p) : _p(p) {}
  mein_iterator& operator++() { if(_p) { _p = _p->next; } return *this;}
  mein_iterator operator++(int) {mein_iterator retval = *this; ++(*this); return retval;}
  bool operator==(mein_iterator other) const {return _p == other._p;}
  bool operator!=(mein_iterator other) const {return !(*this == other);}
  reference operator*() const {return *_p;}
};
```

UNIVERSITÄT BONN

Beispiel

Nun muss unsere Liste noch Iteratoren erzeugen. Es ist üblich die Iteratorklasse innerhalb des Containers zu definieren.

```
class Liste{
public:
  ...

  // Definition der Iteratorklasse innerhalb des Containers
  class mein_iterator : public std::iterator< /* Wie oben */ > {
    /* Wie oben */
  };

  mein_iterator begin() {return mein_iterator(this->head);}
  mein_iterator end()   {return mein_iterator::pointer(nullptr);}
```

Beispiel

Nun kann durch die Liste iteriert werden.

```
#include <datentypen/liste.hpp>

void main() {
  datentypen::Liste l;
  datentypen::Nodeptr current;

  l.insert_front(10);
  current = l.insert_front(30);
  l.insert_front(50);
  // ...

  // Liste wie üblich ausdrucken
  l.print();

  // Liste mithilfe des Iterators ausdrucken
  for(const auto& el: l) {
    std::cout << el.data_ << " -> ";
  }
  std::cout << std::endl;
```

UNIVERSITÄT BONN

For each

In der nächsten Vorlesung diskutieren wir eine Auswahl von iterierenden Funktionen aus `<algorithm>`. Hier ein Vorgeschmack. Um eine Funktion für alle Elemente eines Iteratorbereichs auszuführen, verwendet man `std::for_each`.

```
std::set<int> zahlen({ 2, 6, 1, 3, 9, 4, 5 });
auto drucke_doppeltes = [] (int x) -> void { std::cout << 2*x << std::endl; };

std::for_each  (begin(zahlen), end(zahlen), drucke_doppeltes);
```

Zusammenfassung

Um durch eine Folge von Elementen zu iterieren, verwendet man Iteratoren

Die erstmalige Definition eines eigenen Iterators scheint mühseelig, aber die vermeindliche Mühe erhöht die Codequalität immens

Haben Sie Fragen?