

# SWE Zusammenfassung

Elias Leonhardsberger

7. Juli 2025, Hagenberg

## Inhaltsverzeichnis

<b>1</b>	<b>C++</b>	<b>2</b>
1.1	Standard Library . . . . .	2
1.1.1	Strings . . . . .	2
1.1.2	Ein-/Ausgabe . . . . .	2
1.2	STL . . . . .	4
1.2.1	Iteratoren . . . . .	4
1.2.2	Behälter . . . . .	6
1.2.3	Funktionsobjekte . . . . .	7
<b>2</b>	<b>Java</b>	<b>8</b>
2.1	Standard Library . . . . .	8
2.2	JCF . . . . .	8
<b>3</b>	<b>Softwaremuster</b>	<b>8</b>
3.1	OOP . . . . .	8
3.2	Gang of 4 . . . . .	8
3.3	MVC . . . . .	8
3.4	Iterator . . . . .	8
3.5	Composite . . . . .	8
<b>4</b>	<b>Syntaxvergleich</b>	<b>9</b>
4.1	C++ . . . . .	9
4.1.1	Interface . . . . .	9
4.1.2	Basisklasse . . . . .	9
4.1.3	Abgeleitete Klasse . . . . .	10
4.1.4	Main . . . . .	14
4.2	Java . . . . .	16
4.2.1	Interface . . . . .	16
4.2.2	Basisklasse . . . . .	16
4.2.3	Abgeleitete Klasse . . . . .	16
4.2.4	Main . . . . .	18

# 1 C++

## 1.1 Standard Library

Der Namensraum *std* ist für die Standardbibliothek von C++ und C reserviert. Alle C Header-Dateien sind in C++ verfügbar, ihr Name ist immer mit einem *c* vorangestellt, z.B. *cstdio* für *stdio.h* in C.

### 1.1.1 Strings

Im gegensatz zu C, wo Strings als char-Arrays implementiert sind, gibt es in C++ die Klasse `std::string`, die es ermöglicht mit Templates Strings aus `char`, `wchar_t` oder einem beliebigen eigenen Typ zu erstellen. Die ersten beiden Varianten sind bereits vordefiniert.

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

### 1.1.2 Ein-/Ausgabe

In C und C++ gibt es keine in die Sprache eingebauten Ein-/Ausgabefunktionen. In C werden die Funktionen aus *stdio.h* verwendet, in C++ die Klassen aus *iostream*.

Der Unterschied ist, dass C++ mit Streams arbeitet, diese sind

- typsicher
- implementierbar für eigene Klassen
- effizienter da man nicht auf interpretierte Formatzeichenketten angewiesen ist (z.B. `%d` für `int`)
- auf Zeichenebene Thread-sicher

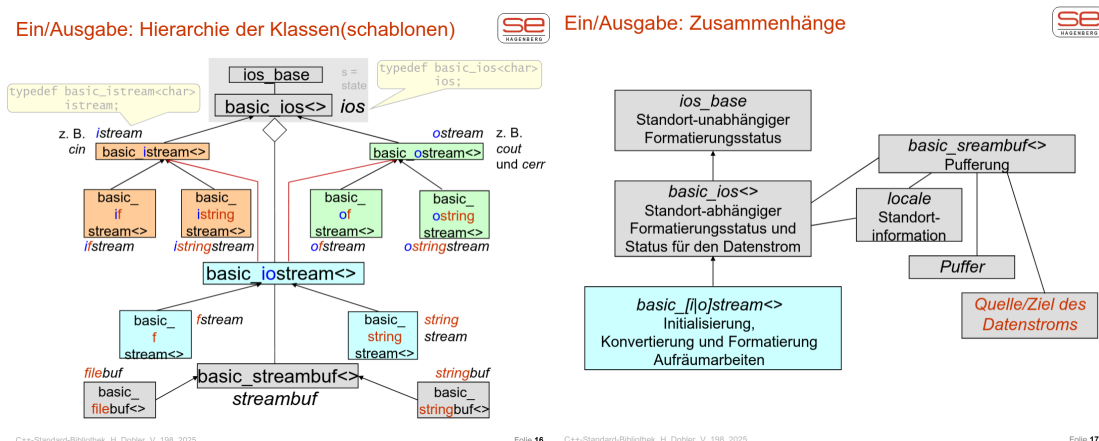


Abbildung 1: Klassenhierarchie der Ein-/Ausgabeklassen in C++

Diese Streams sind in mehreren Header-Dateien aufgeteilt:

- **iosfwd** für Vorwärtsdeklarationen (Sehr klein)

- **streambuf** für die Pufferung von Ein-/Ausgabe
- **istream** für die Eingabe
- **ostream** für die Ausgabe
- **iostream** für die Standard-Ein-/Ausgabe
- **fstream** für Dateiein-/ausgabe
- **sstream** für String-Ein-/Ausgabe
- **strstream** für für Char\*-Ein-/Ausgabe
- **iomanip** für Ein-/Ausgabeformatierung

Flush kann auch mit Stream Aufrufen verkettet werden, z.B. `cout « std::flush; std::endl` flusht den Stream auch nachdem er einen Zeilenumbruch ausgegeben hat. Achtung kann Performanceprobleme verursachen, da es den Puffer jedes Mal leert! Wenn mehrere Zeilenumbrüche ausgegeben werden eher den folgenden Ausschnitt verwenden.

1

```
stream << "\n"
```

Es gibt auch diverse Manipulatoren, die das Verhalten von Streams beeinflussen.

- `std::dec` für Dezimalzahlen
- `std::hex` für Hexadezimalzahlen
- `std::oct` für Oktalzahlen
- `std::setbase(n)` für die Basis der Zahlendarstellung
- `std::setfill(c)` für das Füllzeichen
- `std::setprecision(n)` für die Anzahl der Nachkommastellen
- `std::setw(n)` für die Breite des Feldes

## 1.2 STL

!!Diese Schablone hat keine Laufzeitprüfungen!!

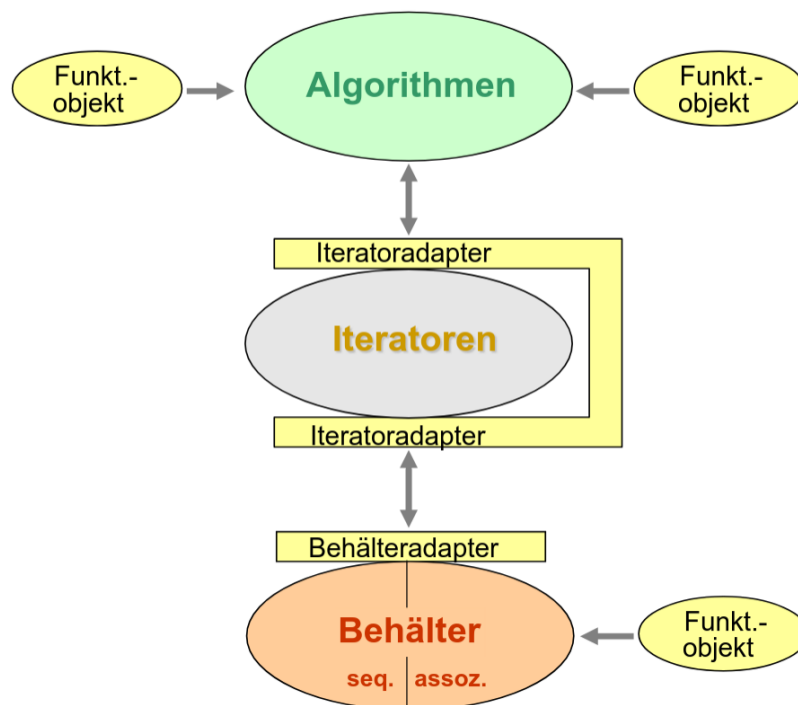
*Doblers Foliensatz*

Generische Programmierung statt OOP um den Code wiederverwendbar und effizient zu machen.

Die STL ist in mehrere Teile aufgeteilt:

- **Behälter** für die Speicherung von Daten
- **Iteratoren** für den Zugriff auf die Daten
- **Algorithmen** für die Verarbeitung von Daten
- **Funktionsobjekte** für die Kapselung von Funktionen für andere Komponenten
- **Adapter** für die Anpassung von Behälter und Iteratoren

### Konzept des Zusammenwirkens



STL, H. Dobler, V. 19, 2025

Folie 6

Abbildung 2: Verhältnisse der STL Komponenten

### 1.2.1 Iteratoren

Iteratoren sind eine Verallgemeinerung von Zeigern, die eine abstrakte Schnittstelle zwischen Behältern und Algorithmen darstellen.

In der STL sind Iteratoren mithilfe von Ducktyping umgesetzt, das heißt, wenn die Schnittstelle erfüllt ist, also die Methoden richtig implementiert sind, dann kann der Iterator verwendet werden.

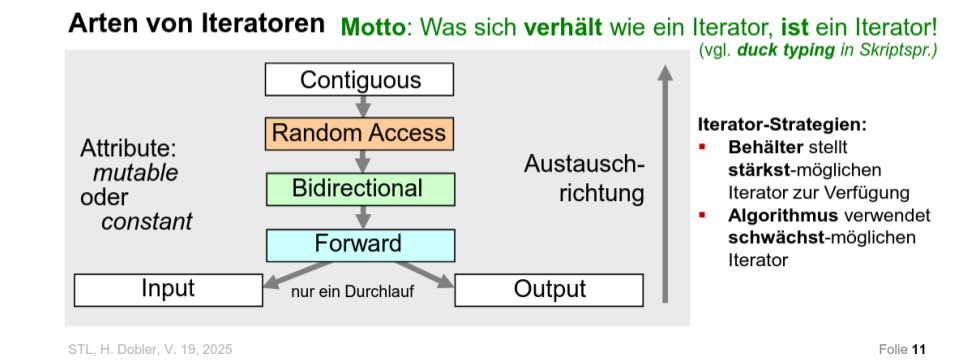


Abbildung 3: Iteratoren in der STL

Allgemein müssen Iteratoren die Vergleichsoperatoren `==` und `!=` implementieren, um zu überprüfen, ob zwei Iteratoren auf das gleiche Element zeigen, und die Dereferenzierungsoperatoren `*`, um auf das Element zuzugreifen.

Der **Vorwärtsiterator** implementiert zusätzlich den Inkrementoperator `++`, um zum nächsten Element zu gelangen.

Der **bidirektionale Iterator** implementiert zusätzlich den Dekrementoperator `-`, um zum vorherigen Element zu gelangen.

Der **Random-Access-Iterator** implementiert zusätzlich die Operatoren `+`, `-`, `+=` und `-=`, um auf ein beliebiges Element im Behälter zuzugreifen, und die Vergleichsoperatoren `<`, `>`, `<=` und `>=`, um die Position der Iteratoren zu vergleichen. Dabei wird auch ein Differenztyp benötigt (z.B. `ptrdiff_t`).

Der **Input-Iterator** ist ein Vorwärtsiterator, der nur gelesen werden kann.

Der **Output-Iterator** ist ein Vorwärtsiterator, auf den nur geschrieben werden kann.

Streams und damit auch Dateien können mit dem `istream_iterator` und `ostream_iterator` als Iteratoren verwendet werden.

Es gibt auch Adapter für Iteratoren, wie

- **reverse\_iterator** für die Umkehrung der Iteration (`v.rbegin()` und `v.rend()`)
- **back\_insert\_iterator** für das Einfügen von Elementen am Ende eines Behälters
- **front\_insert\_iterator** für das Einfügen von Elementen am Anfang eines Behälters
- **insert\_iterator** für das Einfügen von Elementen an einer bestimmten Position in einem Behälter

Die Einfügeiteratoren sind da um die Einfügeoperationen in Algorithmen zu abstrahieren.

Iteratoreigenschaften werden verwendet um verschiedene Implementierungen für verschiedene Iteratoren zu ermöglichen. Die Eigenschaften sind. Außerdem wird der Wertetyp und der Distanztyp damit gespeichert.

### 1.2.2 Behälter

Behälter sind Klassen, die Daten speichern und verwalten. Es gibt verschiedene Arten von Behältern, die sich in ihrer Implementierung und ihrem Verhalten unterscheiden. Die wichtigsten Behältertypen sind:

Behältertyp	Beschreibung	Verwendung
<b>vector</b>	Dynamisches Array, das hinten offen ist und schnellen direkten Zugriff ermöglicht	Standard für sequentielle Daten
<b>list</b>	Doppelt verkettete Liste, die schnelle Einfüge- und Löschoperationen ermöglicht, aber keine schnelle Zugriffsfunktion	Wenn häufige Einfüge-/Löschungenoperationen erforderlich sind
<b>deque</b>	beidseitig offenes dynamisches Array, das schnellen Zugriff auf beide Enden ermöglicht	Wenn häufige Einfüge-/Löschungenoperationen an beiden Enden erforderlich sind, aber direkter Zugriff auch wichtig ist
<b>array</b>	Schnittstelle für statisches Array	Wenn die Größe des Arrays zur Compilezeit bekannt ist und STL Algorithmen verwendet werden
<b>forward_list</b>	Minimale einfach verkettete Liste	Wenn Speicherverbrauch wichtig ist und nur Vorwärtsiteration benötigt wird

Behältertyp	Beschreibung	Verwendung
<b>stack</b>	LIFO-Adapter für einen Behälter. Der genaue Behälter ist nicht spezifiziert, aber meist ein <i>deque</i> oder <i>vector</i>	Wenn nur die zuletzt hinzugefügten Elemente benötigt werden.
<b>queue</b>	FIFO-Adapter für einen Behälter. Der genaue Behälter ist nicht spezifiziert, aber meist ein <i>deque</i> oder <i>list</i>	Wenn nur die zuerst hinzugefügten Elemente benötigt werden.
<b>priority_queue</b>	Adapter für einen Behälter, der die Elemente nach Priorität sortiert. Der genaue Behälter ist nicht spezifiziert, aber meist ein <i>vector</i> oder <i>deque</i>	Wenn die Elemente nach Priorität sortiert werden müssen.
<b>set</b>	Assoziativer Behälter, in dem die Elemente der Zugriffsschlüssel sind. (Zugriff $O(\log n)$ )	Wenn die Elemente als Zugriffsschlüssel dienen.
<b>multiset</b>	Set in dem Duplikate erlaubt sind	Wenn die Elemente als Zugriffsschlüssel dienen aber Duplikate erlaubt sind
<b>map</b>	Assoziativer Behälter, der zu jedem Schlüssel ein Element zuordnet.	Wenn auf die Elemente nach einem Schlüssel zugegriffen wird.
<b>multimap</b>	Map in der Schlüsselduplikate erlaubt sind	Wenn auf die Elemente nach einem Schlüssel zugegriffen wird, aber Duplikate erlaubt sind.

Assoziative Behälter sind entweder mit einem Rot-Schwarz-Baum oder einem Hash-Table implementiert.

### 1.2.3 Funktionsobjekte

## **2 Java**

### **2.1 Standard Library**

### **2.2 JCF**

## **3 Softwaremuster**

### **3.1 OOP**

### **3.2 Gang of 4**

### **3.3 MVC**

### **3.4 Iterator**

### **3.5 Composite**



## 4 Syntaxvergleich

Um den Syntax von C++ und Java zu vergleichen, hab ich ein kleines Beispielprogramm geschrieben mit:

- einem Interface
- einer Basisklasse, die das Interface implementiert
- einer abgeleiteten Klasse, die von der Basisklasse erbt
- einer Iteratorimplementierung in der abgeleiteten Klasse
- einer Main-Methode, die ein Objekt der abgeleiteten Klasse verwendet

### 4.1 C++

#### 4.1.1 Interface

```
1  template <typename T> // Generic programming with templates
2  class interface // in C++ there is no special keyword for interfaces,
   ↪ so we use a class
3  {
4  public:
5      virtual ~interface() = default; // Virtual destructor for proper
   ↪ cleanup of derived classes
6
7      virtual T getValue(int index) const = 0; // Pure virtual function,
   ↪ must be implemented by derived classes
8      virtual T operator[](int index) const = 0; // operator overload for
   ↪ indexing, must be implemented by derived classes
9      virtual void addValue(T value) = 0; // Pure virtual function, must
   ↪ be implemented by derived classes
10     virtual void operator+=(T value) = 0; // operator overload for
   ↪ adding a value, must be implemented by derived classes
11     virtual int getSize() const = 0; // Pure virtual function, must be
   ↪ implemented by derived classes
12 };
```

#### 4.1.2 Basisklasse

```
1  #include "interface.h"
2
3  template <typename T>
4  class baseclass : public interface<T>
5  {
6  private:
7      int _size;
8
9  public:
```

```

10     baseclass(int size = 10) : _size(size) {} // Constructor with default
    ↪ parameter
11
12     virtual int getSize() const override
13     {
14         return _size;
15     }
16 };

```

### 4.1.3 Abgeleitete Klasse

```

1  #include "baseclass.h"
2  #include <iostream>
3
4  template <typename T>
5  class derivedclass : public baseclass<T>
6  {
7      friend std::ostream& operator<<(std::ostream& os, const
    ↪ derivedclass<T>& obj)
8      {
9          for (int i = 0; i < obj._currentIndex; ++i)
10         {
11             os << obj._data[i] << " "; // Access _data directly since
    ↪ it's a friend
12         }
13
14         return os;
15     }
16
17 private:
18     T* _data;
19     int _currentIndex;
20
21 public:
22     // Constructor calls base constructor with size parameter and
    ↪ initializes _currentIndex
23     derivedclass(int size = 10) : baseclass<T>(size), _currentIndex(0)
24     {
25         // Allocate memory for _data based on the size provided
26         _data = new T[size];
27     }
28
29     // Destructor to clean up allocated memory
30     virtual ~derivedclass()
31     {
32         delete[] _data; // Free the allocated memory for _data
33     }
34

```

```

35 // copy constructor to create a deep copy of the derivedclass
36 derivedclass(const derivedclass& other) :
    ↳ baseclass<T>(other.getSize()),
    ↳ _currentIndex(other._currentIndex)
37 {
38     // Allocate memory for _data and copy the values from the other
    ↳ instance
39     _data = new T[other.getSize()];
40     for (int i = 0; i < other.getSize(); ++i)
41     {
42         _data[i] = other._data[i];
43     }
44 }
45
46 // move constructor to transfer ownership of resources
47 derivedclass(derivedclass&& other) noexcept :
    ↳ baseclass<T>(other.getSize())
48 {
49     // Transfer ownership of the data pointer and current index
    ↳ from the other instance
50     _data = other._data;
51     _currentIndex = other._currentIndex;
52
53     // Set the other instance's data pointer to nullptr to avoid
    ↳ double deletion
54     other._data = nullptr;
55     other._currentIndex = 0; // Reset the index of the moved-from
    ↳ instance
56 }
57
58 // copy assignment operator to create a deep copy of the
    ↳ derivedclass
59 derivedclass& operator=(const derivedclass& other)
60 {
61     if (this != &other) // Check for self-assignment
62     {
63         // Clean up existing resources
64         delete[] _data;
65
66         // Allocate new memory and copy the values from the other
        ↳ instance
67         _data = new T[other.getSize()];
68         for (int i = 0; i < other.getSize(); ++i)
69         {
70             _data[i] = other._data[i];
71         }
72
73         _currentIndex = other._currentIndex;

```

```

74     }
75
76     return *this;
77 }
78
79 // move assignment operator to transfer ownership of resources
80 derivedclass& operator=(derivedclass&& other) noexcept
81 {
82     if (this != &other) // Check for self-assignment
83     {
84         // Clean up existing resources
85         delete[] _data;
86
87         // Transfer ownership of the data pointer and current index
88         ↪ from the other instance
89         _data = other._data;
90         _currentIndex = other._currentIndex;
91
92         // Set the other instance's data pointer to nullptr to
93         ↪ avoid double deletion
94         other._data = nullptr;
95         other._currentIndex = 0; // Reset the index of the
96         ↪ moved-from instance
97     }
98
99     return *this;
100 }
101
102 // Override getValue to return the value at the current index
103 virtual T getValue(int index) const override
104 {
105     if (index < 0 || index >= this->getSize())
106     {
107         throw std::out_of_range("Index out of range");
108     }
109
110     return _data[index];
111 }
112
113 // Override operator[] to provide access to the value at the
114 ↪ current index
115 virtual T operator[](int index) const override
116 {
117     return getValue(index); // Use getValue to access the value at
118     ↪ the index
119 }

```

```

116 // Override addValue to add a value at the current index and
    ↪ increment the index
117 virtual void addValue(T value) override
118 {
119     if (_currentIndex < 0 || _currentIndex >= this->getSize())
120     {
121         throw std::out_of_range("Index out of range");
122     }
123
124     _data[_currentIndex] = value; // Store the value at the current
    ↪ index
125     _currentIndex++; // Increment the index for the next value
126 }
127
128 // Override operator+= to add a value at the current index and
    ↪ increment the index
129 virtual void operator+=(T value) override
130 {
131     addValue(value); // Use addValue to handle the addition and
    ↪ index increment
132 }
133
134 class iterator
135 {
136 private:
137     T* _ptr; // Pointer to the data
138     int _index; // Current index in the data array
139
140 public:
141     // Constructor to initialize the iterator with a pointer
142     iterator(T* ptr, int index) : _ptr(ptr), _index(index) {}
143
144     // Overload the dereference operator to return the value at the
    ↪ current index
145     T& operator*() const
146     {
147         return _ptr[_index];
148     }
149
150     // Overload the increment operator to move to the next index
151     iterator& operator++()
152     {
153         _index++;
154         return *this;
155     }
156
157     // Overload the equality operator to compare two iterators
158     bool operator==(const iterator& other) const

```

```

159     {
160         return _index == other._index && _ptr == other._ptr;
161     }
162
163     // Overload the inequality operator to compare two iterators
164     bool operator!=(const iterator& other) const
165     {
166         return !(*this == other);
167     }
168
169 };
170
171 iterator begin()
172 {
173     return iterator(_data, 0); // Return an iterator pointing to
174     ↪ the start of the data
175 }
176
177 iterator end()
178 {
179     return iterator(_data, _currentIndex); // Return an iterator
180     ↪ pointing to the end of the data
181 }
182 };

```

#### 4.1.4 Main

```

1  #include "derivedclass.h"
2  #include <iostream>
3
4  int main()
5  {
6      try
7      {
8          // Create an instance of derivedclass with a size of 5
9          derivedclass<int> myDerived(5);
10
11         // Add some values to the derived class
12         myDerived.addValue(10);
13         myDerived.addValue(20);
14         myDerived += 30;
15
16         // Access values using operator[]
17         std::cout << "Value at index 0: " << myDerived[0] << std::endl;
18         std::cout << "Value at index 1: " << myDerived[1] << std::endl;
19         std::cout << "Value at index 2: " << myDerived.getValue(2) <<
20         ↪ std::endl;

```

```

21      // Demonstrate copy constructor
22      derivedclass<int> copiedDerived(myDerived);
23      std::cout << "Copied: " << copiedDerived << std::endl;
24
25      // Demonstrate move constructor
26      derivedclass<int> movedDerived(std::move(myDerived));
27      std::cout << "Moved: " << movedDerived << std::endl;
28
29      // Iterate through the values using the iterator
30      std::cout << "Iterating through values: ";
31      for (auto i : movedDerived)
32      {
33          std::cout << i << " ";
34      }
35      std::cout << std::endl;
36  }
37  catch (const std::exception& e)
38  {
39      std::cerr << "Exception: " << e.what() << std::endl;
40  }
41
42  return 0;
43 }

```

## 4.2 Java

### 4.2.1 Interface

```
1 package javademo;
2
3 // Java supports interfaces and generics but not operator overloading
4 public interface Interface<T> {
5     T getValue(int index);
6
7     void addValue(T value);
8
9     int getSize();
10    // No operator[] or operator+= in Java
11 }
```

### 4.2.2 Basisklasse

```
1 package javademo;
2
3 public abstract class BaseClass<T> implements Interface<T> {
4     private int size;
5
6     BaseClass(int size) {
7         this.size = size;
8     }
9
10    @Override
11    public int getSize() {
12        return size;
13    }
14
15    // Java supports not implementing interfaces in abstract classes
16 }
```

### 4.2.3 Abgeleitete Klasse

```
1 package javademo;
2
3 import java.util.ArrayList;
4 import java.util.Iterator;
5 import java.util.NoSuchElementException;
6
7 public class DerivedClass<T> extends BaseClass<T> implements
8     ⇨ Iterable<T> {
9     private ArrayList<T> data;
10    private int currentIndex;
```



```

11 public DerivedClass(int size) {
12     super(size);
13     data = new ArrayList<>(size); // generic arrays in Java are
    ↪ semi-supported
14     currentIndex = 0;
15 }
16
17 @Override
18 public T getValue(int index) {
19     if (index < 0 || index >= getSize()) {
20         throw new IndexOutOfBoundsException();
21     }
22
23     return data.get(index);
24 }
25
26 @Override
27 public void addValue(T value) {
28     if (currentIndex < 0 || currentIndex >= getSize()) {
29         throw new IndexOutOfBoundsException();
30     }
31
32     data.add(value);
33     currentIndex++;
34 }
35
36 @Override
37 public String toString() {
38     StringBuilder sb = new StringBuilder();
39
40     for (int i = 0; i < currentIndex; i++) {
41         sb.append(data.get(i));
42         sb.append(" ");
43     }
44
45     return sb.toString().trim();
46 }
47
48 // Iterable implementation for for-each loop
49 @Override
50 public Iterator<T> iterator() {
51     return new Iterator<T>() {
52         private int index = 0;
53
54         @Override
55         public boolean hasNext() {
56             return index < currentIndex;
57         }

```

```

58
59         @Override
60         public T next() {
61             if (!hasNext()) {
62                 throw new NoSuchElementException();
63             }
64
65             return data.get(index++);
66         }
67     };
68 }
69
70 // No operator overloading in Java, so no operator[] or operator+=
71 // No need for destructors or manual memory management
72 }

```

#### 4.2.4 Main

```

1  package javademo;
2
3  public class Demo {
4      public static void main(String[] args) {
5          DerivedClass<Integer> myDerived = new DerivedClass<>(5);
6          myDerived.addValue(10);
7          myDerived.addValue(20);
8          myDerived.addValue(30);
9
10         System.out.println("Value at index 0: " +
11             ↳ myDerived.getValue(0));
12         System.out.println("Value at index 1: " +
13             ↳ myDerived.getValue(1));
14         System.out.println("Value at index 2: " +
15             ↳ myDerived.getValue(2));
16
17         System.out.println("toString: " + myDerived.toString());
18
19         System.out.print("Iterating through values: ");
20         for (int value : myDerived) {
21             System.out.print(value + " ");
22         }
23     }
24 }

```