Informatik I

Prof. Dr. Christian Pape

Abstrakte Datentypen

Inhalt

- Geheimnisprinzip
- Abstrakte Datentypen (ADT)
- Java Interfaces
- Java Collection Framework

Geheimnisprinzip

- Nicht alle Details einer Implementierung preisgeben
  - Interne Datenstrukturen geheim halten (private Attribute)
  - Nur wenige öffentliche Methoden
- Stack
  - Intern: Feld und ein Index für das oberste Element

Stack
-stack : int []   -oben : int
+push(int z) : void   +pop() : int

- Verschiedene Implementierungen von Stack möglich
  - Stack wächst von unten (Feldindex 0) nach oben
  - Stack wächst von oben nach unten

Geheimnisprinzip

- Nicht alle Details einer Implementierung preisgeben
  - Interne Datenstrukturen geheim halten (private Attribute)
  - Nur wenige öffentliche Methoden
- Stack
  - Intern: Feld und ein Index für das oberste Element

Stack
-stack : int []
-oben : int
+push(int z) : void
+pop() : int

0	1	2					7	8
5	2							

oben = 1, bei push(7) wird oben um 1 erhöht

Stack wächst von unten nach oben (links nach rechts)

0	1	2					7	8
							2	5

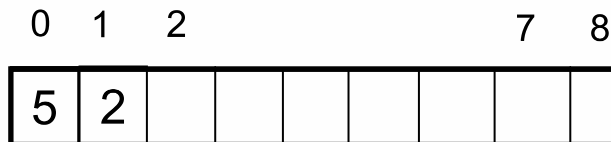
oben = 7, bei push(7) wird oben um 1 reduziert

Stack wächst von oben nach unten (rechts nach links)

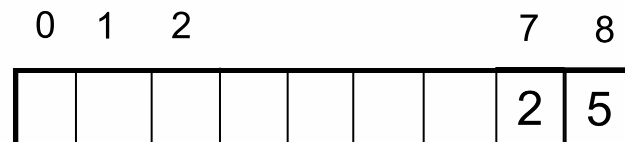
Geheimnisprinzip

```
public class StackA {

  private int [] stack = new int[1000];

  private int oben = 0;

  public void push(int z) {
    stack[oben++] = z;
   }

  public int pop() {
    return stack[--oben];
  }
}
```

```
public class StackB {

  private int [] stack = new int[1000];

  private int oben = stack.length-1;

  public void push(int z) {
    stack[oben--] = z;
   }

  public int pop() {
    return stack[++oben];
  }
}
```



oben = 2, bei push(7)
wird oben um 1 erhöht



oben = 6, bei push(7)
wird oben um 1 reduziert

Geheimnisprinzip

- Vorteil Geheimnisprinzip
  - Programme, welche Stack verwenden müssen nicht geändert werden, wenn nur die internen Details der Implementierung geändert werden
  - Privaten Methoden / Attribute für Benutzer der Klasse nicht von Interesse
  - „Beste" Stackimplementierung für einen Anwendungszweck wählbar
  - **Jede** Stackimplementierung muss sich gleich verhalten
  - Die **Definition** dieses Verhalten definiert einen **abstrakten Datentyp (ADT)**
- In diesem Fall
  - Nur Verhalten von push und pop relevant
  - Definition des Verhaltens mathematisch (Formale Spezifikation) möglich oder umgangssprachlich (Nicht formale Spezifikation)

Stack
+push(int z) : void +pop() : int

Inhalt

- Geheimnisprinzip
- Abstrakte Datentypen (ADT)
- Java Interfaces
- Java Collection Framework

ADT (mathematisch)

Stack
+push(int z) : void +pop() : int

- Stack definiert eine Menge S von Stackobjekten
- Methoden sind Funktionen, die Stackobjekte S ändern

ADT (mathematisch)

Stack
+push(int z) : void +pop() : int

- Stack definiert eine Menge S von Stackobjekten
- Methoden sind Funktionen, die Stackobjekte S ändern

ADT

$$push: \quad S \times Z \to \quad S$$

$$pop: \quad S \to \quad S \times Z$$

$$pop(push(s,z)) = (s,z)$$

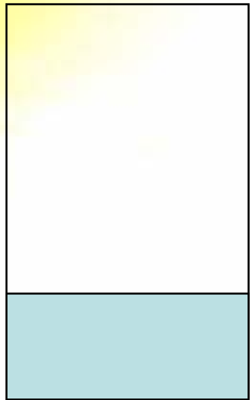
$$push(pop(s)) = s$$

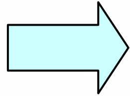
Abstrakte Datentyp Stack

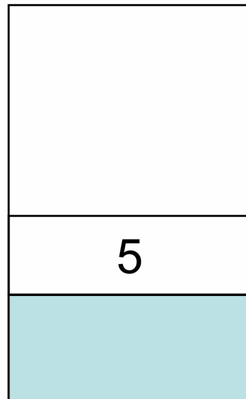
- s.push(z) ändert den Stack s
- s.pop() ändert den Stack s und gibt das oberste Element zurück

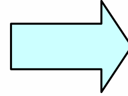
- Implizite Definitionen
  - Alle Funktionen pop und push sowie Mengen S und Z, die diese Bedingungen erfüllen, verhalten sich wie ein Stack
  - Es fehlen noch Fehlerfälle / Leerer Stack

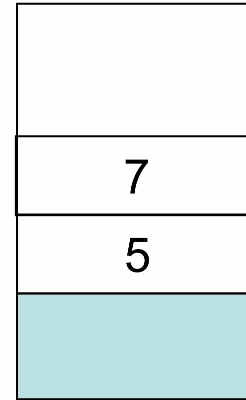
ADT (mathematisch)



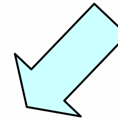


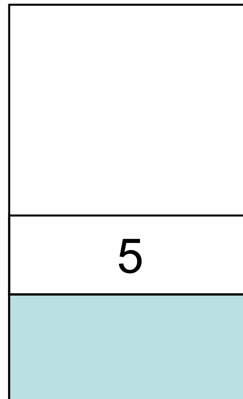






s $\quad push(s,5))$ $\quad push(push(s,5),7)$





$$pop(push(push(s,5),7)) = (push(s,5),7)$$

ADT (nicht formal)

Stack
+push(int z) : void +pop() : int

- Stack definiert eine Menge S von Stackobjekten
- Methoden sind Funktionen, die Stackobjekte S ändern

ADT

```
/**
 * Legt die Zahl z zuoberst auf den Stack
 */
public void push(int z);

/**
 * Gibt die oben auf dem Stack liegende Zahl zurück
 * und entfernt Sie vom Stack, so dass die darunter liegende Zahl
 * das oberste Element des Stacks ist.
 */
public int pop();
```

Inhalt

- Geheimnisprinzip
- Abstrakte Datentypen (ADT)
- Java Interfaces
- Java Collection Framework

Java Interface

- Interface
  - Enthält Deklaration öffentlicher Attribute und Methoden
  - Keine Implementierung enthalten (auch kein Konstruktor)
  - Definiert einen ADT
  - public kann weggelassen werden (andere Modifier nicht erlaubt)

```
public interface Stack {
  /**
   * Legt die Zahl z zuoberst auf den Stack
   */
  public void push(int z);

  /**
   * Gibt Zahl oben auf dem Stack liegende Zahl zurück
   * und entfernt Sie vom Stack, so dass die darunter liegende Zahl nun
   * das oberste Element des Stacks ist.
   */
  public int pop();
}
```

Java Interface

- Interface
  - Definiert eine Java-Typ (wie eine Klasse)
  - Kann für Deklaration von Variablen / Parameter verwendet werden

```
public void macheWas(Stack stack) {
  stack.push(12);
  stack.push(7);
  System.out.println( stack.pop() );
  stack.push(83);
}
```

```
Stack s;
// Fehlt: initialisiere s

s.push(17);
macheWas(s);
int x = s.pop();
```

Java Interface

- Implementierung eines Stack
  - Klasse muss alle Methoden von Stack implementieren
  - Klasse kann noch weitere Methoden / Attribute besitzen
  - Programmierer ist verantwortlich für die korrekte Implementierung

```
public class StackA  implements Stack {

  private int [] stack = new int[1000];

  private int oben = 0;

  public void push(int z) {
   stack[oben++] = z;
  }

  public int pop() {
   return stack[--oben];
  }
}
```

```
public class StackB implements Stack {

  private int [] stack = new int[1000];

  private int oben = stack.length-1;

  public void push(int z) {
   stack[oben--] = z;
  }

  public int pop() {
   return stack[++oben];
  }
}
```

Java Interface

- Verwendung
  - Variable mit konkreter Implementierung eines Stacks
  - Variable lediglich als Subtyp Stack deklarieren (Polymorphie)
  - Compiler prüft, dass nur Methoden des Interfaces verwendet werden
- Zur Laufzeit
  - es ändert sich ansonsten nichts
  - Methoden push, pop von StackA werden aufgerufen
- Vorteil
  - Grosse Programmteile hängen nicht mehr von einer konkreten Implementierung eines Stacks ab: bessere Wartbarkeit der Programme

```
public void macheWas(Stack stack) {
  stack.push(12);
  stack.push(7);
  System.out.println( stack.pop() );
  stack.push(83);
}
```

```
Stack s;
s = new StackA();

s.push(17);
macheWas(s);
int x = s.pop();
```

Java Interface

- Interface
  - Eine Klasse kann mehr als ein Interface implementieren
  - Eine einfache Form von „Mehrfachvererbung“

```
public class StackA  implements Stack, Comparable  {

 // Implementierung Stack wie zuvor

 /**
  * a negative integer, zero, or a positive integer as
  *  this object is less than, equal to, or greater than
  *  the specified object.
  */
  public int compareTo(Object o) {
     // Implementierung fehlt
  }

}
```

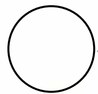
```
AStack as =  new StackA();
Comparable c = as;
Stack s1 = as;
Stack s2 = new StackB();

macheWas(s1);
macheWas(s2);

if ( c.compareTo(s2) == 0) {
  // s1 ist gleich s2 …
}
```

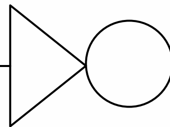
Interfaces in UML

Interfaces

- Interface kann von Interface erben
- Implementierte Klasse muss dann all diese Interfaces imlementieren

ComparableStack **Comparable**





push(int z) : void compareTo(o : Object) : int
pop() : int

```
public class StackA  implements ComparableStack  {

  // Implementierung Stack wie zuvor

public int compareTo(Object o) {
     // Implementierung fehlt
  }

}
```

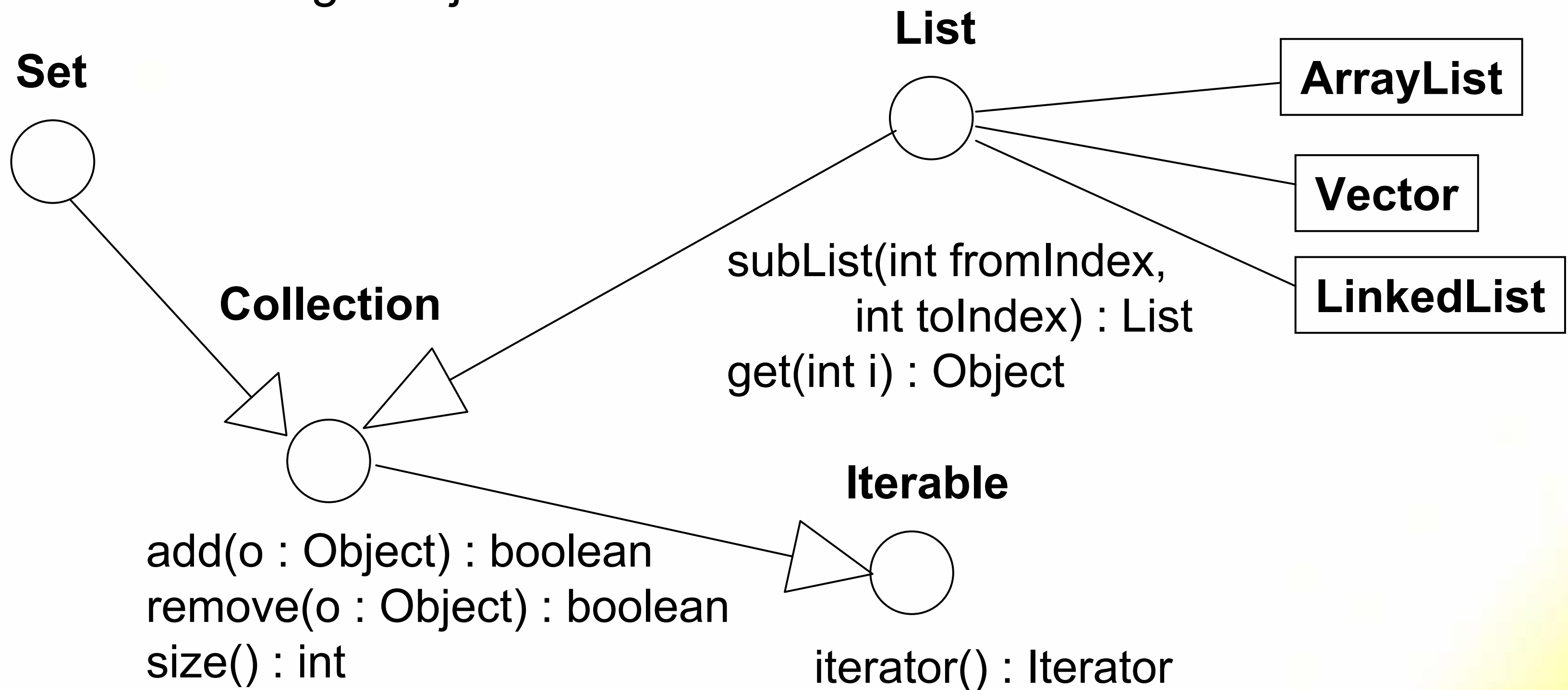
```
Stack a1 =  new StackA();
Comparable c = new StackA()

if ( c.compareTo(a1) == 0) {
  // beide Stacks sind gleich
}
```

Inhalt

- Geheimnisprinzip
- Abstrakte Datentypen (ADT)
- Java Interfaces
- Java Collection Framework

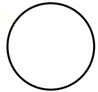
Java Collections Framework

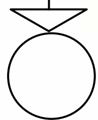
- http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html
- Sammlung von ADTen und deren Implementierungen
- Kleiner Auszug aus java.util :



Java Collections Framework

List





Collection

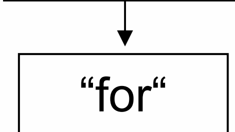
add(o : Object) : boolean
remove(o : Object) : boolean
size() : int

- Listen haben ein Anfang und ein Ende
- add() fügt o am **Ende** an
- remove() entfernt **erste Vorkommen** von o ab Anfang der Liste

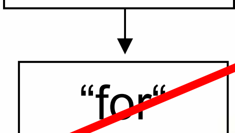
List list = // …. List erzeugen

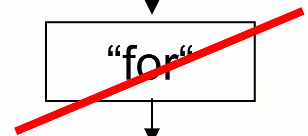
list.add(“Thanks“);

list.add(“ for“);

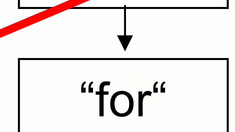


list.add(“for“);





list.remove(“for“)



Java Collection Framework

```
public boolean suchen(List list, Comparable c) {
   for (Iterator iterator = list.iterator();
             iterator().hasNext(); // noch ein Element vorhanden
        ) {
      Object o = iterator.next(); // gibt nächstes Element zurück
      if ( c.compareTo(o) {
        return true;
      }
   }
   return false;
}
```

JDK 1.5

```
public boolean suchen(List list, Comparable c) {
   for (Object o : list) {
      if ( c.compareTo(o) == 0 ) {
        return true;
      }
   }
   return false;
}
```

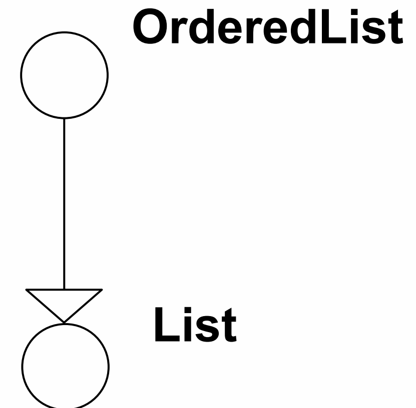
Java Collection Framework

- Methode suchen nutzt keine speziellen Merkmale von List oder Collection
- Best passendste Interface von List verwenden: Iterable
- Methode funktioniert dann für List, Set, …

```
public boolean suchen(Iterable iterable, Comparable c) {
    for (Object o : iterable) {
        if ( c.compareTo(o) == 0 ) {
          return true;
        }
    }
    return false;
}
```

Java Collection Framework

- Verbesserte Suche bei sortierte Listen möglich
- Derartiges Interface existiert allerdings nicht, nur OrderedSet, OrderedMap
- Eigenes erstellen
- Javadoc des Interfaces muss den neuen ADT genau erklären:
  - iterator() gibt Element in „natürlicher“ Ordnung zurück
  - add / remove erhalten die Reihenfolge
  - add(int i) erzeugt einen Fehler, falls Ordnung nicht mehr gewährleistet
  - …




```
public boolean suchen(OrderedList list, Comparable c) {
   for (Object o : list) {
      if ( c.compareTo(o) == 0 ) {
        return true;
      } else if (c.compareTo(o) < 0 ) {
        return false;
      }
   }
   return false;
}
```

Java Collection Framework

```
List list = new ArrayList();
list.add(“Fröhliche“);
list.add(“Weihnachten“);
list.add(“und“);
list.add(“guten“);
list.add(“Rutsch“);

if (suchen(list, “und“)) { // String implementiert Comparable
  // „und“ gefunden
}

Collections.sort(list); // list muss Comparables enthalten

int index = Collections.binarySearch(list, “und“);
                                 // Comparables nötig
if (index > -1) {
  System.out.println(“‘und‘ an Position “ + index + “ gefunden.“);
}
```

Java Collection Framework

- Zeitaufwand binarySearch hängt von Implementierung der verwendeten List ab
- Javadoc-Auszug zur Methode:

This method runs in log(n) time for a "random access" list
(which provides near-constant-time positional access).
If the specified list does not implement the RandomAccess interface and is large, this method will do an iterator-based binary search that performs O(n) link traversals and O(log n) element comparisons.

- RandomAccess enthält keine Methoden – es ist nur ein Interface zur Markierung, dass get(int i) konstanten Zeitaufwand hat
- ArrayList ist eine derartige Implementierung
- LinkedList nicht: $O(n + \log n) = O(n)$

Java Collection Framework

- Aufgabe
  - Implementieren Sie ein Interface Stack, um Werte von Typ Object zu verwalten.
  - Implementieren Sie einen Stack mit Hilfe einer Liste: ListStack.
  - Der Stack sollte einen Konstruktor haben, bei dem die Instanz von Typ List übergeben wird.
  - Im Konstruktor muss die Liste leer gemacht werden.
  - Sehen Sie sich die Javadoc von List für eine Beschreibung der Methoden an.