04-Objekte-2-Objektstrukturen

Objektorientierte Programmierung | Matthias Tichy

Software Engineering
Programming Languages

universität ulm

Lernziele

- Objektstrukturen
- Vererbung
- Liskovsches Substitutionsprinzip
- Code Clones
- Autoboxing / Unboxing
- Generics

Beispiel: Stack

> Stack
>
> „The Stack class represents a last-in-first-out (LIFO) stack of objects.”

- Methoden:
  - push: Objekt oben auf den Stack legen
  - pop: oberstes Objekt vom Stack entfernen
  - peek: oberstes Objekt auf dem Stack zurückliefern
  - size: Anzahl der Objekte auf dem Stack

https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Stack.html

Erzeugung (Instanziierung von Objekten)

```
private static Stack createStack() {
   Stack s = new Stack();
   s.push(12);
   s.push(14);
   s.push(42);
   return s;
}
```

https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Stack.html

Hands-On
Code Stack

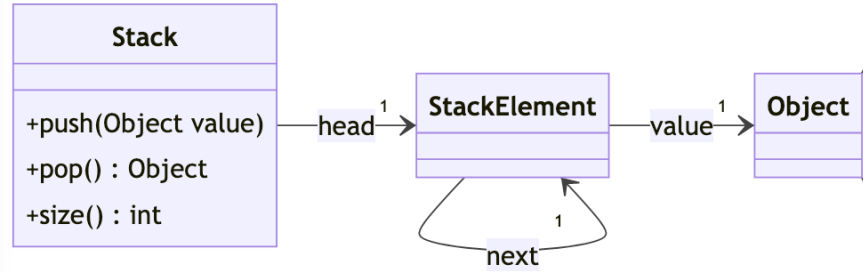
- Live-Objektstruktur: push(12) → push(14) → push(42)

Klassendiagramme

Graphische Darstellung der Klassenstruktur objektorientierter Programme

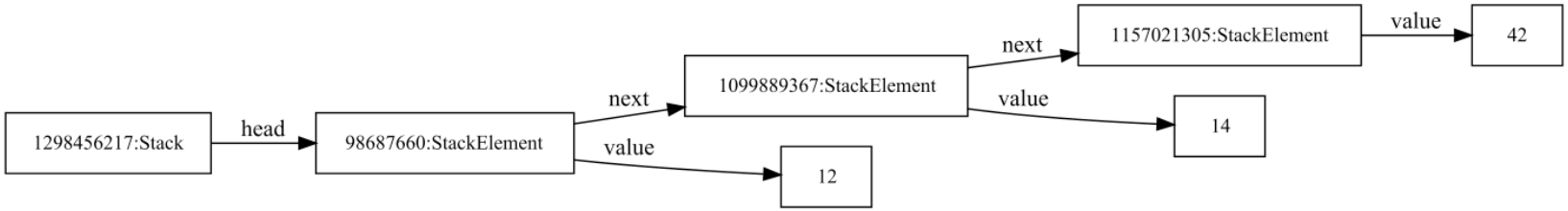
```
public class Stack {
  private StackElement head;
  public Object pop() {...}
  public void push(Object value) {...}
  public int size() {...}
  ...
}
```

```
class StackElement {
  private Object value;
  private StackElement next;
  ...
}
```

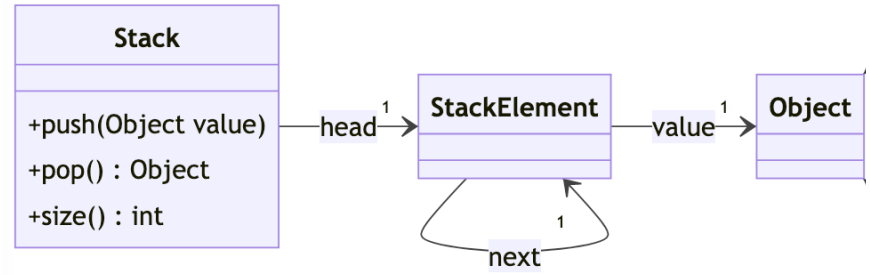


Objektdiagramm / Instance Specification

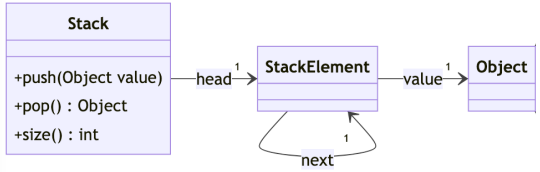
Graphische Darstellung der Objektstruktur

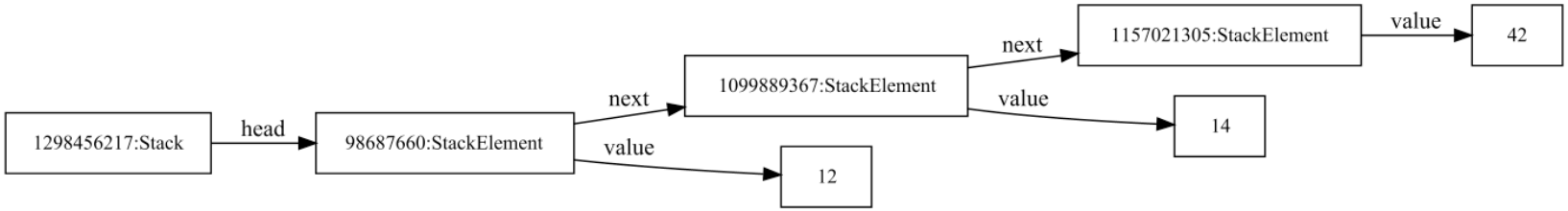


- Zugehöriges Klassendiagram:



Referenztypen: null-Value



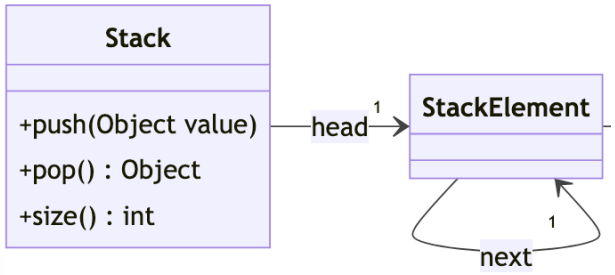


- Welchen Wert hat `1157021305.next`?
  → `null`
- Referenzentypen können neben einer Referenz auch den Wert null haben
- Sie referenzieren auf „nichts“.
- Welchen Wert hat `1157021305.next.value`?
- Es gibt keinen Wert, sondern eine NullPointerException
- → Programmabbruch (später mehr zu Exceptions).

Erzeugung (Instanziierung von Objekten)

```
private static Stack crea
    Stack s = new Stack();
    s.push(12);
    s.push(14);
    s.push(42);
    return s;
}
```

```
public void push(Object value) {
  // empty stack
  if (head == null) {
    StackElement elem = new StackElement(value);
    head = elem;
  } else {
    StackElement elem = head;
    // find end
    while (elem.getNext() != null) {
      elem = elem.getNext();
    }
  elem.setNext(new StackElement(value));  }
}
```



Hands-On
Code Stack (2)

- Implementierung der Methode (push)
- Tests

Beispiel: List

List

„An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list."

- Methoden, u.a.:
  - add: Objekt ans Ende der Liste hinzufügen
  - remove: Objekt aus der Liste entfernen
  - get: Objekt an einer Position
  - size: Anzahl der Objekte in der Liste

https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/List.html

Hands-On
Code LList

- Implementierung der Methoden (add)
- Live-Objektstruktur

Redundanz der Implementierungen

Stack

- Methoden:
  - push
  - pop
  - peek
  - size
- Implementierung
  - Menge von StackElementen
  - head-StackElement für erstes Element
  - Verknüpfung über next-Attribut

List

- Methoden:
  - add
  - remove
  - get
  - size
- Implementierung
  - Menge von ListElementen
  - head-ListElement für erstes Element
  - Verknüpfung über next-Attribut

- „Code Clones“ → Wiederverwendung und Ergänzung statt Redundanz

Vererbung

Vererbung bei Objektorientierten Programmiersprachen erlaubt eine Wiederverwendung von Eigenschaften (Methoden und Attribute) einer vererbenden Klasse (Superklasse/supertype) durch eine erbende Klasse (Subklasse/subtype).

Die erbende Klasse kann das Verhalten ergänzen bzw. anpassen.

Aufruf der Methode in der Superklasse

```
public class InheritingStack extends LList {
  public void push(Object value) {
    super.add (value);
  }
  //[…]
}
```

Widening Reference Conversion

> „A *widening reference conversion* exists from any reference type S to any reference type T, provided S is a subtype of T“

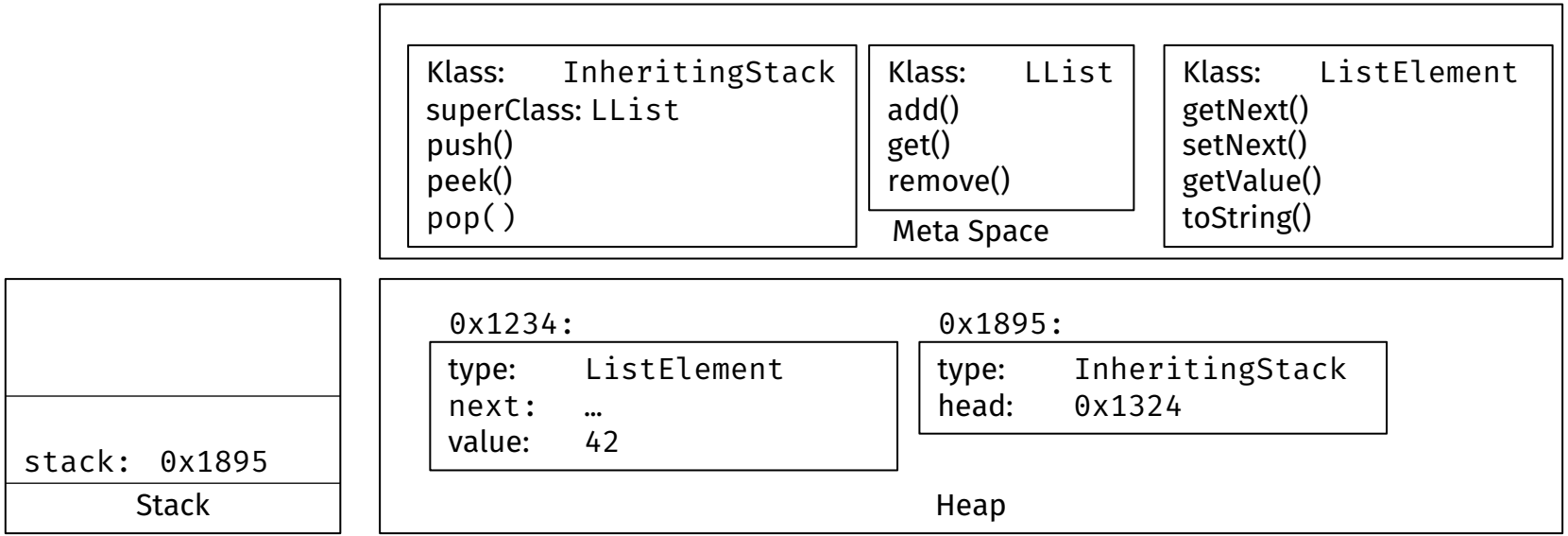
```
InheritingStack stack = new InheritingStack();
stack.add(42)
stack.push(12)
LList stack2 = new InheritingStack();
// list is for the compiler just an object conforming to LList
stack2.add(42)
~~stack2.push(12)~~
```

```
public class InheritingStack extends LList {
  //[…]
}
```

The Java Language Specification, Java SE 19 Edition, §5.1.5

Methodenaufrufe und Vererbung

```
InheritingStack stack = new InheritingStack();
stack.add(42)
stack.push(12)
```



Liskovsches Substitutionsprinzip

Subtype Requirement:

> „Let $\phi(z)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.”

- → Wenn ein Nutzer ein Objekt einer Klasse T erwartet, sollte sich ein Objekt einer Klasse S (und S ist Subklasse von T) für den Nutzer in Bezug auf eine Eigenschaft „gleich“ verhalten.

Barbara H. Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16, 6 (Nov. 1994), 1811–1841. https://doi.org/10.1145/197320.197383

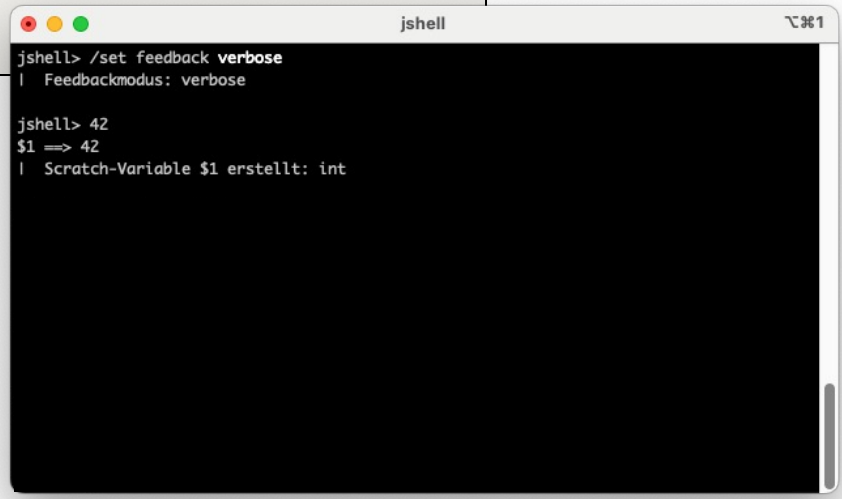
Hands-On
Code InheritingStack

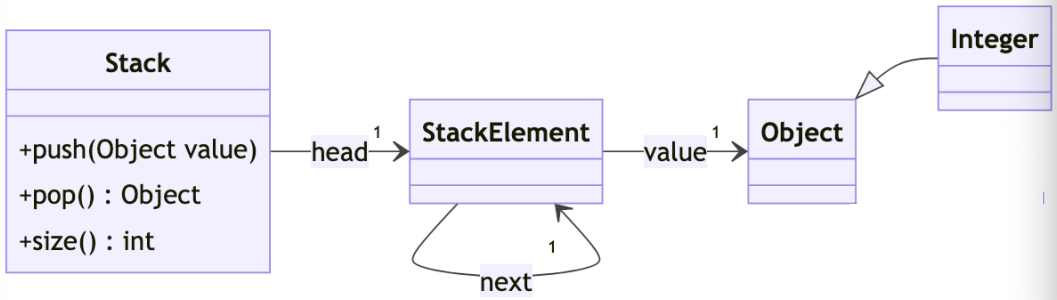
- Implementierung der Methoden von `InheritingStack`
- Tests: `ParameterizedListsTests`

Klassen ←/→ Primitive Datentypen

```
private static Stack createStack() {
    Stack s = new Stack();
    s.push(12);
    s.push(14);
    s.push(42);
    return s;
}
```

```
public void push(Object value) {
    ...
}
```





Klassen ←/→ Primitive Datentypen

- Primitive Datentypen sind keine Klassen!
- Wrapper-Klassen zur Nutzung von primitiven Datentypen in Objektstrukturen

  → automatische Konvertierung durch den Compiler (Autoboxing / Unboxing)

- Wrapper-Klassen bieten weitere Funktionalitäten (z.B. parsing)

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Stack in JAVA SE 19 - Vererbungshierarchie

Stack in JAVA SE 19 – Implementierung

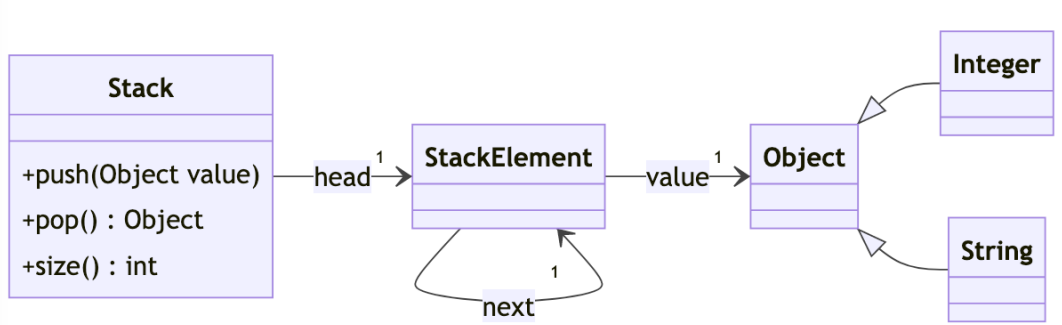
```
public E push(E item) {
        addElement(item);
        return item;
    }
```

```
public synchronized E peek() {
        int len = size();
        if (len == 0)
            throw new EmptyStackException();
        return elementAt(len - 1);
    }
```

```
public synchronized E pop() {
        E       obj;
        int     len = size();
        obj = peek();
        removeElementAt(len - 1);
        return obj;
    }
```

Generics

```
Stack s = new Stack();
s.push("Ulm");
s.push(12);
```




```
Integer v1 = s.pop();
```

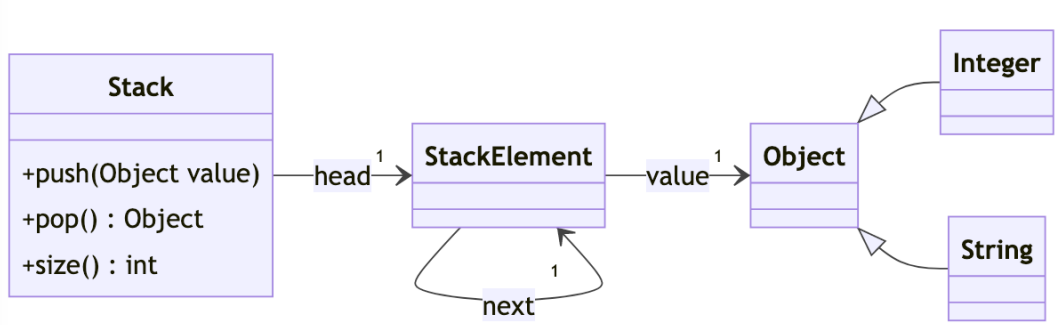
Type mismatch: cannot convert from Object to Integer

```
String v2 = s.pop();
```

Type mismatch: cannot convert from Object to String

- String und Integer und alle anderen Klassen erben von Object
- Unser Stack kann alles abspeichern
- Der Compiler kann nicht prüfen/einschränken, was drin ist

Generics




```
Stack s = new Stack();
s.push("Ulm");
s.push(12);
```

```
Integer v1 = (Integer) s.pop();

Integer v2 = (Integer) s.pop();
```

Type Cast. „Compiler: das ist ein Integer … hoffentlich“.

Upps…

- Man muss immer den Type Cast schreiben → Typprüfung abgeschaltet
- Bei Typfehlern zur Laufzeit → `ClassCastException`

Generics

Type Parameter → Generische Implementierung

```
public class GenericsStack<T> {
  private GenericsStackElement<T> head;
  public T pop() {...}
  public void push(T value) {...}
  public int size() {...}
}
```

```
class GenericsStackElement<T> {
  private T value;
  private GenericsStackElement<T> next;
  GenericsStackElement (T value) {...}
  T getValue() {...}
  GenericsStackElement<T> getNext(){...}
  void setNext(GenericsStackElement<T> next) {...}
}
```

- Für den Typ Parameter T muss bei der Anwendung ein konkreter Typ angegeben werden.

Generics

Argumente für den Typ Parameter

```
GenericsStack<Integer> s = new GenericsStack<Integer>();
s.push(12);
s.push("Ulm");
```

The method push(Integer) in the type GenericsStack<Integer> is not applicable for the arguments (String)

```
Integer v1 = s.pop();

String v2 = s.pop();
```

Type mismatch: cannot convert from Integer to String

Optional

Reale Implementierung des GenericsStack

```
public class GenericsStack<T> {
  private GenericsStackElement<T> head;
  public Optional<T> pop() {...}
  public void push(T value) {...}
  public int size() {...}
}
```

Optional

“A container object which may or may not contain a non-null value.” / MayBe

```
public class GenericsStack<T> {
  private GenericsStackElement<T> head;
  public Optional<T> pop() {...}
  public void push(T value) {...}
  public int size() {...}
}
```

```
public final class Optional<T> {
  private final T value;
  public boolean isPresent() {...}
  public boolean isEmpty() {...}
  public T get() {...}
  ...
}
```

- „Zwingt“ den Entwickler beide Fälle (empty+present) zu beachten
- Vermeidung von NullPointerExceptions

https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/Optional.html

Optional

“A container object which may or may not contain a non-null value.” / MayBe

```
Stack s = new Stack();
s.push(12);

var element = s.pop();
if (element.isPresent()) {
  Integer v1 = (Integer) element.get();
  ...
} else {
  ...
}
```

- „Zwingt“ den Entwickler beide Fälle (empty+present) zu beachten
- Vermeidung von NullPointerExceptions

https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/Optional.html

Optional

“A container object which may or may not contain a non-null value.” / MayBe

```
Stack s = new Stack();
s.push(12);

var element = s.pop();
Integer v1 = element.orElse(42); // Default mitgeben, falls isEmpty()==true
```

- Weitere Methoden, siehe JavaDoc
- insb. interessant bei funktionalem Java (später)

https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/Optional.html
https://www.baeldung.com/java-optional

Optional in TypeScript/JavaScript

```
interface Stack {
  head:StackElement;
}
interface StackElement {
  value:any,
  next?:StackElement,
}
const s:Stack = {head: {value: 12, next: {value: "Ulm"}}}
console.log(s);
console.log(s.head.value);
console.log(s.head.next?.value);
console.log(s.head.next?.next?.value);
```

- Spezielle Syntax für Optional-Chaining

https://www.typescriptlang.org/play?#code/JYOwLgpgTgZghgYwgAgMpkQa2QbwLABQyyAFhHACYBc6WAogDYQC2E4A3IQL6GiSyIUtBJkYs2Y
XIWIA3OAwCuEKnBABPADTTkICAA8wAfhoYRY1uC0EeBBAHsQAZ0mOTWZAF4pRUuWrfiWXklKmQARgAmK0CdfTBQnDlFZWQAIgBVBmZUm2IbL
mRCeyc7JgA6BjsAcwAKRwBKTlsHR1KICuq6srJKMqSlRqKWto7ax26-Mt0DQz7giEHmkvLKsYne6aMpuNn+haagA

Lernziele

- Objektstrukturen
- Vererbung
- Liskovsches Substitutionsprinzip
- Code Clones
- Autoboxing / Unboxing
- Generics