Universität Stuttgart

PSE:
Lambdas und funktionale Programmierung in Java

Prof. Dr.-Ing. Steffen Becker

Vorlesung 22


Universität Stuttgart
Recap
PSE:
Qualitätssicherung: Testen
Prof. Dr.-Ing. Steffen Becker
Vorlesung 21

Definition Testen

Testen ist die experimentelle Überprüfung eines konkreten Softwaresystems hinsichtlich seiner (funktionalen und nicht-funktionalen) Qualität.

Die Experimente basieren auf **Testfällen**.

Ermittlung von Testfällen

Eine **automatische** Ableitung der optimalen Testsuiten für jeden Fall ist **nicht** möglich.

Das Ziel ist es daher, gute **Heuristiken** zu finden, um leistungsfähige Tests abzuleiten.

”be the enemy of your code“.

Arten von Tests

Black-Box:
nur aus der **Spezifikation** abgeleitete Testfälle (z.B. aus Vor- und Nachbedingungen).

White-Box (Glass-Box):
aus dem **Programmcode** abgeleitete Testfälle.

Domänentest (Forts.)

Repräsentative Prüfung von

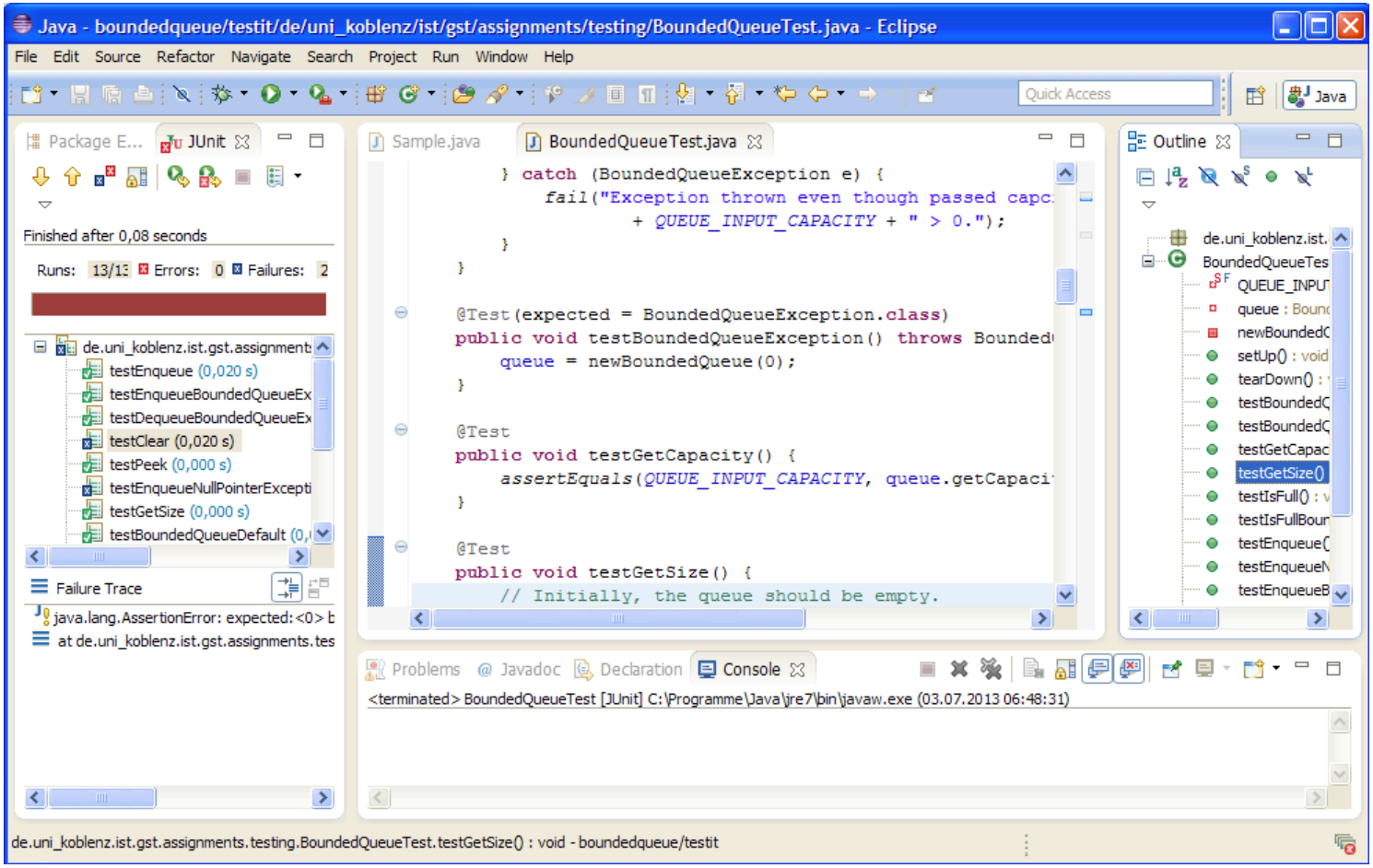
- einigen **Normalfällen**
- Möglichst viele **Grenzfälle** (**Extremfälle**)
- einige **Fehlerfälle**

Davon sind die **Extremfälle** (leere Eingabe, nur identische Daten, maximale Anzahl von Daten, etc.) die wichtigsten.

Glass-Box-Testen

Beim **Glasbox-Testen** (strukturiertes Testen, Whitebox-Testen) werden die Testdaten auf Basis des **Programmcodes** bzw. der Systemstruktur erstellt.

Die Testdaten sind bewusst so aufgebaut, dass **ein ausgewählter Kontrollflusspfad** im Programm durchlaufen wird.


Java - boundedqueue/testit/de/uni_koblenz/ist/gst/assignments/testing/BoundedQueueTest.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Package E...
JUnit
Finished after 0,08 seconds
Runs: 13/13
Errors: 0
Failures: 2
de.uni_koblenz.ist.gst.assignment
testEnqueue (0,020 s)
testEnqueueBoundedQueueEx
testDequeueBoundedQueueEx
testClear (0,020 s)
testPeek (0,000 s)
testEnqueueNullPointerExcepti
testGetSize (0,000 s)
testBoundedQueueDefault (0,
Failure Trace
java.lang.AssertionError: expected:<0> b
at de.uni_koblenz.ist.gst.assignments.tes
Sample.java
BoundedQueueTest.java
} catch (BoundedQueueException e) {
fail("Exception thrown even though passed capc
+ QUEUE_INPUT_CAPACITY + " > 0.");
}
}
@Test(expected = BoundedQueueException.class)
public void testBoundedQueueException() throws Bounded
queue = newBoundedQueue(0);
}
@Test
public void testGetCapacity() {
assertEquals(QUEUE_INPUT_CAPACITY, queue.getCapaci
}
@Test
public void testGetSize() {
// Initially, the queue should be empty.
Outline
de.uni_koblenz.ist.
BoundedQueueTes
QUEUE_INPUT
queue : Bound
newBoundedQ
setUp() : void
tearDown() :
testBoundedQ
testBoundedQ
testGetCapac
testGetSize()
testIsFull() :
testIsFullBoun
testEnqueue(
testEnqueueN
testEnqueueB
Problems
Javadoc
Declaration
Console
<terminated> BoundedQueueTest [JUnit] C:\Programme\Java\jre7\bin\javaw.exe (03.07.2013 06:48:31)
de.uni_koblenz.ist.gst.assignments.testing.BoundedQueueTest.testGetSize() : void - boundedqueue/testit

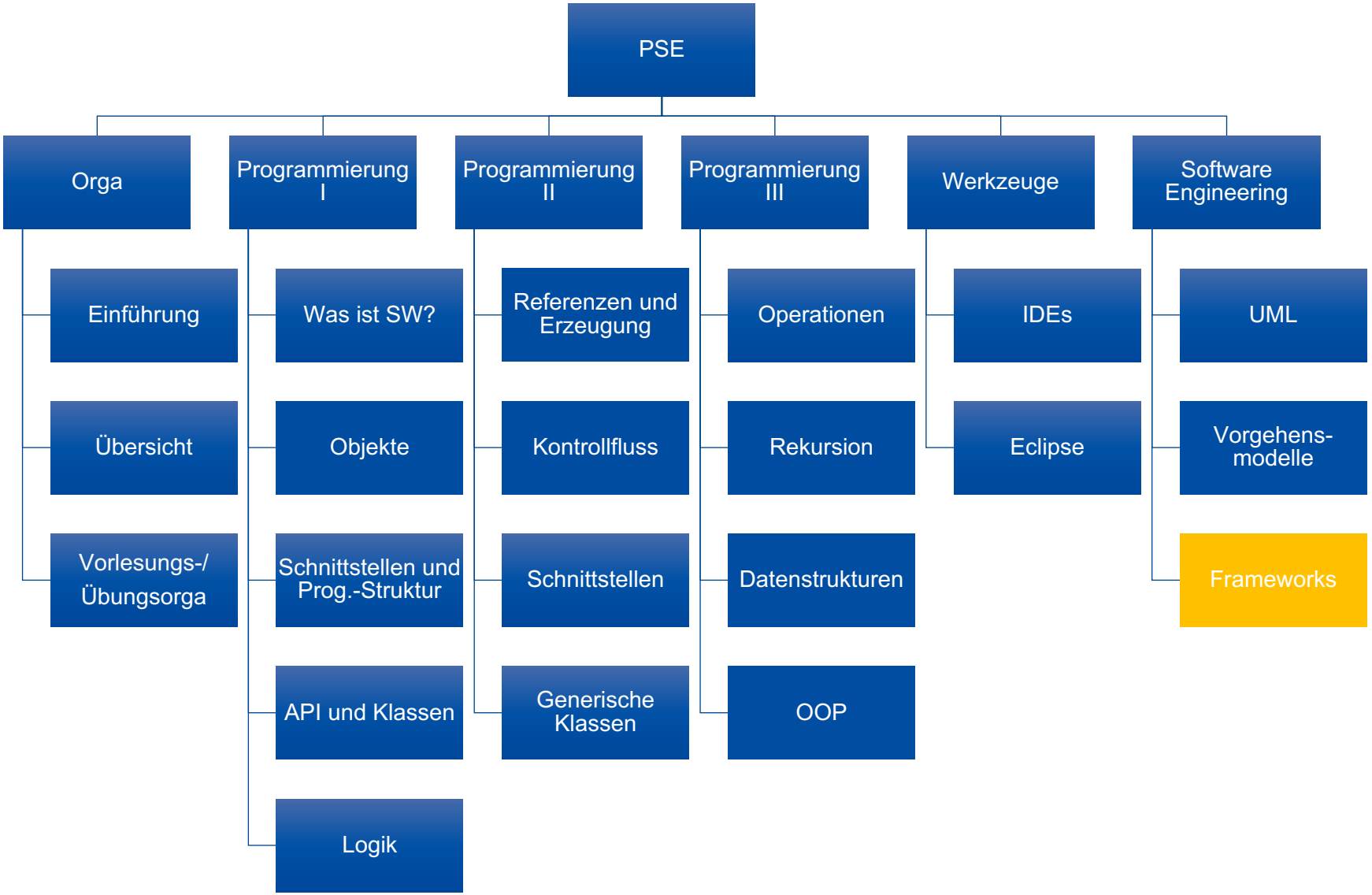
Universität Stuttgart

PSE:
Lambdas und funktionale Programmierung in Java

Prof. Dr.-Ing. Steffen Becker

Vorlesung 22

Vorlesungsübersicht



Lernziele dieser VL

- Funktionen und funktionale Programmierung als eigenständiges Konzept der Programmierung
- Übergabe von Funktionen als Argumente für Operationen
- (Anonyme) Innere Klassen als Konzept vor Java 1.8 zur Realisierung von Lambda-Ausdrücken
- Lambda-Ausdrücke und funktionale Schnittstellen als Realisierung in Java 1.8
- Definition und Benutzung von Lambda-Ausdrücken in Java
- Syntax- und Stilregeln im Umgang mit Lambda-Ausdrücken

Hauptquellen:
- Java ist auch eine Insel, Kapitel 11

Literatur für diese Vorlesung

Dieses Slide-Deck basiert weitgehend auf dem Buch „Java ist auch eine Insel“ (Kapitel 11: *Lambda-Ausdrücke und funktionale Programmierung*)

http://openbook.rheinwerk-verlag.de/javainsel/11_001.html

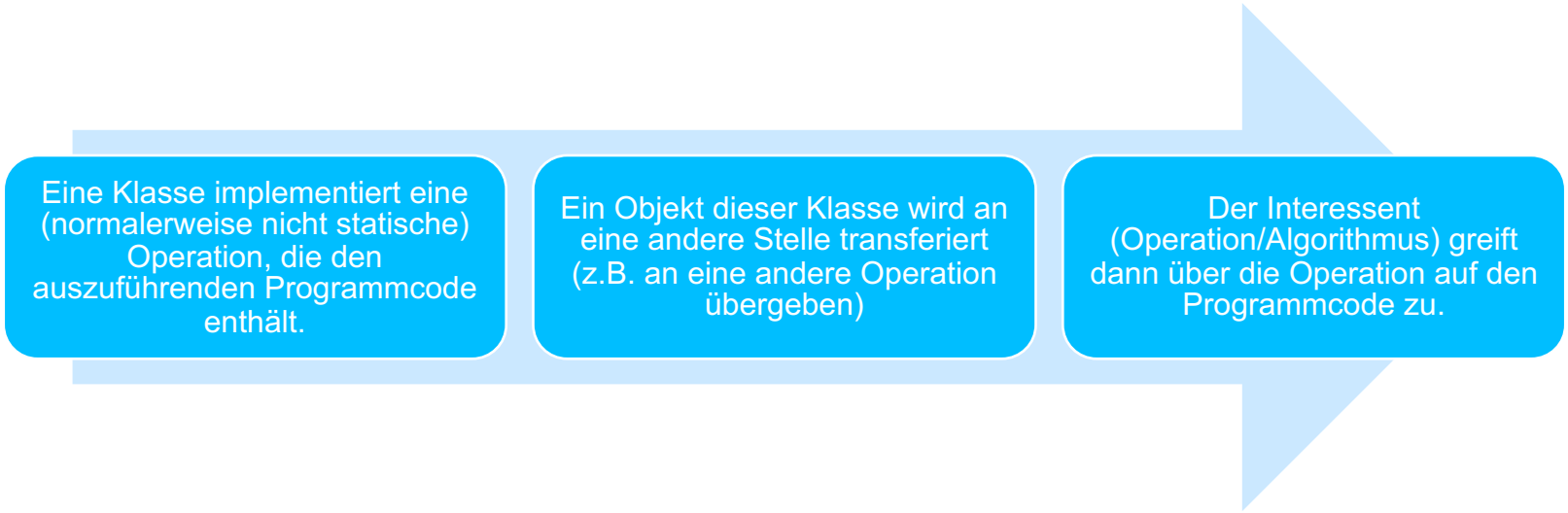
Didaktik dieser VL

- Vortragen der Informationen
- Erklären an Beispielen

Code = Daten

- Bisher haben wir hauptsächlich **Daten** an Operationen übergeben, z.B. Zahlen, Strings und Objekte
- Java-Code ist auch Daten
- Daher sollte es in der Lage sein, Java-Code genau wie Daten zu übergeben
- Dies erlaubt es, das Verhalten der Algorithmen anzupassen
- Typische Java-Beispiele
- Code, der von einem Thread ausgeführt werden soll, wird als `Runnable`-Objekt übergeben (bietet eine Operation `run()`)
- Die Operation `java.util.Timer.schedule` erwartet einen `TimerTask`
- Die Operation `java.util.Collections.sort` erwartet einen `Comparator`

Muster des Transports von Code in Java mit Objekten



Diesen Mechanismus werden wir uns nun in verschiedenen Varianten näher ansehen.

Verschiedene Möglichkeiten, den zu übergebenden Code zu kapseln

Beispiel: Comparator-Schnittstelle in Java

Method Detail

compare

```
int compare(T o1,
            T o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

The implementor must ensure that `sgn(compare(x, y)) == -sgn(compare(y, x))` for all `x` and `y`. (This implies that `compare(x, y)` must throw an exception if and only if `compare(y, x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `((compare(x, y)>0) && (compare(y, z)>0))` implies `compare(x, z)>0`.

Finally, the implementor must ensure that `compare(x, y)==0` implies that `sgn(compare(x, z))==sgn(compare(y, z))` for all `z`.

It is generally the case, but *not* strictly required that `(compare(x, y)==0) == (x.equals(y))`. Generally speaking, any comparator that violates this condition should clearly indicate this fact. The recommended language is "Note: this comparator imposes orderings that are inconsistent with equals."

Parameters:

`o1 - the first object to be compared.`

`o2 - the second object to be compared.`

Returns:

`a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.`

Throws:

`NullPointerException - if an argument is null and this comparator does not permit null arguments`

`ClassCastException - if the arguments' types prevent them from being compared by this comparator.`

Beispiel: Sortierung abgeschnittener Strings – Verwendung von Lambdas (seit Java 1.8)

```java
import java.util.*;

public class CompareTrimmedStrings {

    public static void main(String[] args) {

        String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };

        Arrays.sort(words, (String s1, String s2) -> {
            return s1.trim().compareTo(s2.trim());
        });

        System.out.println(Arrays.toString(words));

    }
}
```

Code übergeben als ein **Lambda**-Ausdruck (in fetter Schrift)

Der Java-Begriff "Lambda-Ausdruck" geht auf den Lambda-Kalkül (auch als λ Kalkül geschrieben) aus den 1930er Jahren zurück und ist eine formale Sprache für die Untersuchung von Funktionen.

Inferierter Typ des Lambda Ausdrucks

```java
package de.unistuttgart.iste.rss;

import java.util.*;

public class CompareTrimmedStrings {
    public static void main(final String[] args) {
        final String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };
        final Comparator<String> typedComparator =
           (final String s1, final String s2) -> {
                return s1.trim().compareTo(s2.trim());
            };
        final Comparator<String> typedComparator2 = (s1, s2) -> {
            return s1.trim().compareTo(s2.trim());
        };
        final var stringComparator = typedComparator;
        stringComparator.compare("Test1", "Test2");
        Arrays.sort(words, typedComparator2);
    }
}
```

2. Beispiel: Filterung

```
public class LambdaTest {
    private final static List<String> words =
            Arrays.asList("Hallo", "Welt", "ich", "bin", "da");


    public static List<String> captialWords() {
        final List<String> result = new LinkedList<String>();
        for (final String word : words) {
            if (Character.isUpperCase(word.charAt(0))) {
                result.add(word);
            }
        }
        return result;
    }
    public static List<String> lowercaseWords() {
        final List<String> result = new LinkedList<String>();
        for (final String word : words) {
            if (Character.isLowerCase(word.charAt(0))) {
                result.add(word);
            }
        }
        return result;
    }
}
```

Lambdas einführen

Identifizieren Sie im Code auf dem vorherigen Dia den Code-Klon und vermeiden Sie ihn mit Hilfe von Lambdas.

Wie müssen Sie den Code unter Verwendung von Lambdas ändern, um den Code-Klon zu vermeiden?

Zeit: 10min

Ergebnis: Online-Diskussion

2. Beispiellösung: Filterung

```java
@FunctionalInterface
interface Predicate<G> {
    boolean isMatch(G value);
}


public class LambdaTest2 {
    private final static List<String> words =
            Arrays.asList("Hallo", "Welt", "ich", "bin", "da");
    private final static Predicate<String> upperCase =
           (s) -> Character.isUpperCase(s.charAt(0));
    private final static Predicate<String> lowerCase =
           (s) -> Character.isLowerCase(s.charAt(0));

    public static void main(final String[] args) {
        filterWords(upperCase);
        filterWords(lowerCase);
    }

    public static List<String> filterWords(final Predicate<String> filter) {
        final List<String> result = new LinkedList<String>();
        for (final String word : words) {
            if (filter.isMatch(word)) {
                result.add(word);
            }
        }
        return result;
    }
}
```

Java-interne Realisierung von Lambdas

Exkursion: innere Klassen

Einfaches Beispiel

- Modellierung eines Bankkontos und der darauf ausgeführten Operationen (z.B. Abhebung, Überweisung, etc.):
  Eine Klasse Account (Konto), eine Klasse Action (Aktion)
- Die Aktion sollte eng an den Account gekoppelt sein (kein Action-Objekt ohne passendes Account-Objekt)

Idee

- Platziere die Klasse Action in die Klasse Account
- Action ist eine innere Klasse (aka. member class)

In Java:

- Die innere Klasse in den Rahmen ihrer äußeren Klasse stellen

Exkursion Beispiel: Account- und Action-Klassen

```
package lecture;

public class Account {
  private int accountNumber;
  private int balance;
  private Action lastAction;


  public class Action {
    private String action;
    private int sum;
    Action (String name, int sum) {
      this.action = name;
      this.sum = sum;
    }
    public String toString() {
      return accountNumber + ":" + action + sum;
    }
  }


  public void withdraw (int sum) {
    balance = balance – sum;
    lastAction = new Action(”withdrawal", sum);
  }
  // weitere, z.B. Einzahlen
}
```

Action ist innere Klasse des Account - sie wird im Bereich des Account deklariert

Zugriff auf ***privates (!)*** Feld der äußeren Klasse

Exkursion: innere Klassen

- Innere Klassen dürfen keine statischen Eigenschaften haben (vgl. statische innere Klassen)
- Innere Klassen ähnlich wie Attribute von Klassen:
  - Objekte von inneren Klassen existieren nur dann, wenn ein Objekt der äußeren Klasse existiert.
- Objekte der inneren Klasse haben Zugriff auf **alle** Teile ihres äußeren Klassenobjekts.
- Objekte von inneren Klassen haben eine implizite Referenz auf das Objekt ihrer äußeren Klasse:
  - Innerhalb des Codes von Action bezieht sich `this` auf das aktuelle Action-Objekt und `Account.this` auf das äußere Account Objekt
- Oftmals als anonyme Klassen verwendet (siehe Beispiel unten)

Ein sehr häufiger Anwendungsfall sind Listener in der GUI-Programmierung

Beispiel: Sortierung abgeschnittener Strings – Verwendung von inneren Klassen

```
import java.util.*;

public class CompareTrimmedStrings {

    public static void main(String[] args) {

        class TrimmingComparator implements Comparator<String> {

            @Override
            public int compare(final String s1, final String s2) {
                return s1.trim().compareTo(s2.trim());
            }
        }

        String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };

        Arrays.sort(words, new TrimmingComparator() );

        System.out.println(Arrays.toString(words));
    }
}
```

Zu übergebender Code (hier: innere Klasse; könnte auch eine öffentliche Klasse sein)

Code wird "transportiert".

Beispiel: Sortierung abgeschnittener Strings – Verwendung von inneren Klassen

```
import java.util.*;

public class CompareTrimmedStrings {

    public static void main(String[] args) {

        String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };

        Arrays.sort(words, new Comparator<String>() {

            @Override
            public int compare(final String s1, final String s2) {
                return s1.trim().compareTo(s2.trim());
            }
        });

        System.out.println(Arrays.toString(words));

    }
}
```

Code, der als Objekt einer **anonymen inneren** Klasse übergeben wird

Lambda-Ausdrücke: Richtlinien für die Verwendung

Operationsdeklarationen vs. Lambda-Ausdrücke

Operationsdeklaration

```
public int compare
( String s1, String s2 ){ return s1.trim().compareTo( s2.trim() ); }
```

Lambda Ausdruck

```
( String s1, String s2 ) ->
{ return s1.trim().compareTo( s2.trim() ); }
```

Allgemeine Syntax eines Lambda Ausdrucks:

```
'(' LambdaParameter ')' '->' '{' Anweisungen '}'
```

Funktionale Schnittstellen

Nicht jede Schnittstelle kann in einem Lambda-Ausdruck verwendet werden, und es gibt eine Schlüsselbedingung, wann ein Lambda-Ausdruck verwendet werden kann.

Interfaces, die nur eine Operation (abstrakte Methode) haben, werden als funktionale Interfaces bezeichnet. Eine abstrakte Klasse mit genau einer abstrakten Methode zählt nicht als funktionales Interface.

Java (>8) kommt mit vielen Schnittstellen, die als funktionale Schnittstellen gekennzeichnet sind. Darüber hinaus führt Java 8 mit dem Paket java.util.function mehr als 40 neue funktionale Schnittstellen ein. Beispiele:

- `interface Runnable { void run(); }`
- `interface Supplier<T> { T get(); } (Java 8)`
- `interface Consumer<T> { void accept(T t); } (Java 8)`
- `interface Comparator<T> { int compare(T o1, T o2); }`
- `interface ActionListener { void actionPerformed(ActionEvent e);`

Annotation @FunctionalInterface

- Um anzuzeigen, dass ein Interface als funktionales Interface gedacht ist, existiert im Paket java.lang der Annotationstyp `FunctionalInterface`.
- Damit ist gekennzeichnet, dass es bei genau einer abstrakten Operation und damit bei einem funktionalen Interface bleiben soll.

```
@FunctionalInterface
public interface MyFunctionalInterface {
    void foo();
}
```

Abkürzungen von Lambda-Ausdrücken

Wie Operationen haben Lambda-Ausdrücke mögliche Parameter- und Rückgabewerte. Die Java-Grammatik für Lambda-Ausdrücke stellt einige syntaktische Abkürzungen zur Verfügung.

Abkürzung 1: Typinferenz

- Lange Version:

```
Comparator<String> c =
(String s1, String s2) -> { return s1.trim().compareTo( s2.trim() ); };
```

- Abgekürzte Version:

```
Comparator<String> c = (s1, s2) -> { return s1.trim().compareTo( s2.trim() ); };
```

Wenn der Compiler die Typen automatisch ableiten kann, sind Typdefinitionen optional.

Abkürzungen von Lambda-Ausdrücken (Fortsetzung)

Abkürzung 2: Lambda-Körper ist entweder Einzelausdruck oder Block

- Wenn der Körper eines Lambda-Ausdrucks nur aus *einem einzigen Ausdruck* besteht, kann eine verkürzte Notation die Blockklammern und das Semikolon sparen

Lange Version

```
( Lambda parameter ) -> { return expression; }
```

Kurze Version

```
( LambdaParameter ) -> Expression
```

Beispiele

Lange Version	Kurze Version
(s1, s2) -> { return s1.trim().compareTo(s2.trim()); }	(s1, s2) -> s1.trim().compareTo(s2.trim())
(a, b) -> { return a + b; }	(a, b) -> a + b
() -> { System.out.println(); }	() -> System.out.println()

Abkürzungen von Lambda-Ausdrücken (Fortsetzung)

Abkürzung 3: Einzelbezeichner statt Parameterliste und Klammern

- Besteht die Parameterliste nur aus einem einzigen Bezeichner und ist der Typ durch Typinferenz eindeutig, können die runden Klammern weggelassen werden.
- Beispiele

Lange Version	Abgeleiteter Typ	Vollständige Abkürzung
(String s) -> s.length()	(s) -> s.length()	s -> s.length()
(int i) -> Math.abs(i)	(i) -> Math.abs(i)	i -> Math.abs(i)

Methodenreferenzen

Eine **Methodenreferenz** ist eine Referenz auf eine Operation (Methode)
ohne deren Aufruf.
Syntaktisch trennen zwei Doppelpunkte den Klassennamen oder
die Referenz links vom Methodennamen rechts.

Ohne Referenz:

```
Arrays.sort( words, (String s1, String s2) -> StringUtils.compareTrimmed(s1, s2) );
```

Mit einer Referenz:

```
Arrays.sort( words, StringUtils::compareTrimmed );
```

Beispiele mit funktionalen Schnittstellen aus java.util.function

Prädikat

```
Predicate<Character> isDigit = c -> Character.isDigit( c ); // short: Character::isDigit
System.out.println( isDigit.test('a') ); // false
```

```
Predicate<Character> isDigit = Character::isDigit;
List<Character> list = new ArrayList<>( Arrays.asList( 'a', '1' ) );
list.removeIf( isDigit );
```

Funktion

```
Function<Double, Double> abs = a -> Math.abs( a ); // alternatively Math::abs
System.out.println( abs.apply( -12. ) ); // 12.0
```

Beispiele mit funktionalen Schnittstellen aus java.util.function

BiConsumer

```
Map<String, Integer> map = new HashMap<>();
map.put( "Manila", 25 );
map.put( "Leipzig", -5 );
map.forEach( (k,v) -> System.out.printf("%s: %d degrees%n", k, v) );
```

Zusammenfassung

Was sollten Sie nun können?

- Lambda-Ausdrücke lesen und schreiben können
- Funktionale Interfaces lesen und schreiben können
- Anonyme und/oder innere Klassen lesen und schreiben können
- Anonyme und/oder innere Klassen in Lambda-Ausdrücke umwandeln können (und umgekehrt)
- Methodenreferenzen benutzen können
- Syntax- und Stilregeln für Lambda-Ausdrücke anwenden können

Welche API sollten Sie nun nutzen können?

- Annotationen für Lambda-Ausdrücke
  - `@FunctionalInterface`
- Elementare funktionale Schnittstellen der Java-BCL kennen
  - `Predicate`
  - `Function`
  - `BiConsumer`
  - `Comparator`

Universität Stuttgart

PSE:
Streams

Prof. Dr.-Ing. Steffen Becker

Vorlesung 23