

Grundlagen der Programmierung VL 19: Funktionale Programmierung

Prof. Dr. Samuel Kounev,
Jóakim von Kistowski,
Norbert Schmitt

Überblick für heute

- Impliziter Iterator in Schleifen
- Programmierparadigmen
- Funktionale Programmierung
- Lambdas

Impliziter Iterator

DIE DOPPELPUNKT-NOTATION

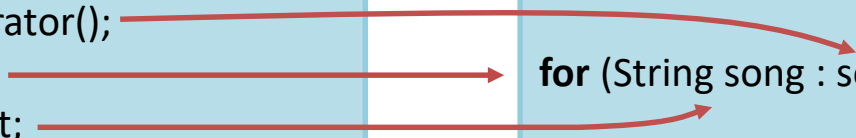
- Iteratoren werden benutzt um Collections in Schleifen zu durchlaufen.
- `hasNext ()` überprüft auf ein nächstes Element
- `next ()` gibt das nächste Element zurück

```
public String getAllSongs(ArrayList<String> songs) {  
    String result = "";  
  
    Iterator e = songs.iterator();  
    while (e.hasNext()) {  
        String song = e.next;  
        result = result + song + "\n";  
    }  
    if (result.equals(""))  
        return ("Keine Lieder vorhanden");  
    else  
        return result;  
}
```

- Vereinfachte Notation für die Schleife
- Iterator wird durch einen Doppelpunkt (:) ersetzt
- Schleife wird zu einer for-Schleife

```
String result = "";  
  
Iterator e = songs.iterator();  
while (e.hasNext()) {  
    String song = e.next;  
  
    result = result + song + "\n";  
}
```

```
String result = "";  
  
for (String song : songs) {  
  
    result = result + song + "\n";  
}
```



- Funktioniert mit Collections und Arrays

Herangehensweisen an Programmiersprachen

PROGRAMMIERPARADIGMEN

- Fundamentale Programmierstile
- Logische Prinzipien hinter der Organisation einer Programmiersprache
- Schließen einander nicht aus
- Einige typische Paradigmen:
 - Imperativ
 - Deklarativ
 - Prozedural
 - Objektorientiert
 - Ereignisorientiert
 - Logisch
 - Funktional

- Imperative Programmierung:
 - Abfolge von Befehlen, die der Computer auszuführen hat
 - Compiler arbeitet Befehle ab, übersetzt in Maschinenbefehle
 - Vergleichbar mit: Möbelbau-Anleitung
 - Sprachen: Assembler, C, Java, ...
- Deklarative Programmierung:
 - Menge von Definitionen, Relationen und Regeln
 - Ergeben einen Bedingen Raum über die Eingabewerte
 - Befehle sind implizit aus den Relationen und Regeln extrahierbar
 - Vergleichbar mit: Mathematischen Formeln
 - Sprachen: SQL, Prolog, ...

- Imperativ
 - Zerlege Anweisungsfolgen in Teilfolgen
 - Anweisungsfolgen sind nach Außen durch Schnittstellen definiert
 - Teilfolgen können sich untereinander aufrufen
 - Parameter erlauben das Weiterreichen von Informationen zwischen Anweisungsfolgen
 - Spezialfall: Rückgabewert ermöglicht die Definition von Prozeduren als Funktion

- Sprachen:
 - Fortan
 - C
 - COBOL
 - ...

- Objekte instanziiieren Objekt-Templates (Klassen)
 - Haben Eigenschaften (Attribute) und können Aktionen ausführen (Methoden)
- Objekte werden häufig zur Modularisierung des Codes verwendet.
- Methoden werden häufig nach dem prozeduralen Paradigma definiert.
- Sprachen:
 - Java
 - C#
 - C++
 - ...

- Sammlung von Ereignisbehandlungsroutinen
- Routinen werden automatisch aufgerufen, wenn das passende Ereignis auftritt. Können andere Ereignisse auslösen.
- Gut mit objektorientierten Sprachen vereinbar
- Typischer Anwendungsfall: GUI-Programmierung
- Sprachen:
 - Visual Basic
 - Pascal
 - ...

- Deklarativ
 - Beruht auf mathematischer Logik
 - Menge von Axiomen
 - Interpreter berechnet Lösungsaussage einer Anfrage aus den Axiomen
- Beispiel in natürlicher Sprache:

Fakten:

Lucia ist die Mutter von Minna.

Lucia ist die Mutter von Klaus.

Minna ist die Mutter von Nadine.

Regel:

Falls *X ist die Mutter von Y* **und** *Y ist die Mutter von Z* **Dann** *X ist die Großmutter von Z.*

Frage/Ziel:

Wer ist die Großmutter von Nadine?

Antwort des Computers, Folgerung aus den Fakten und Regeln:

Lucia

- Bekannteste Sprache: Prolog

FUNKTIONALE PROGRAMMIERUNG

- Deklarativ
 - Benutzt ausschließlich mathematische Funktionen
- Reine Funktionale Programmierung:
 - Verbietet Konzepte aus anderen Paradigmen
 - Variablen
 - Schleifen
 - Objekte
 - Anweisungsfolgen im Allgemeinen
 - ...
- Rekursion ist ein wichtiges Werkzeug!
- Ansprechen von Datenstrukturen ist oft anders gelöst, damit sie besser im rekursiven Kontext verwendet werden können.
- Sprachen:
 - Haskell
 - Erlang
 - Scala
 - ...

- Mathematisch

$$f(n) = \begin{cases} 0, n = 0 \\ 1, n = 1 \\ f(n-1) + f(n-2), \text{sonst} \end{cases}$$

- Imperativ (Java/C)

```
int fib(int order) {  
    if (order == 0) {  
        return 0;  
    } else if (order == 1) {  
        return 1;  
    }  
    return fib(order-1)+fib(order-2);  
}
```

- Funktional (Pseudocode)

```
f(0) = 0  
f(1) = 1  
f(n) = f(n-1) + f(n-2)
```

- (Haskell)

```
fib :: Integer -> Integer  
fib 0 = 0  
fib 1 = 1  
fib n = fib(n-1) + fib(n-2)
```

- Funktionen können als Parameter übergeben werden.
- Ermöglicht Implementierung von Funktionen mit undefinierten Unterfunktionen/Operatoren
 - Beispiel: Berechnung des kürzesten Abstands zwischen zwei Punkten mit variabler Norm des Differenzvektors

- Beispiel:

```
f(g(x,y),x,y) = g(x,y)/x  
sum(x,y) = x + y  
product(x,y) = x * y
```



```
f(sum,2,4) = 3  
f(product,2,4) = 4
```

- Anwendungsfälle: Filterung, Aggregation, Faltung, ...

- Genau wie Anonyme Klassen in Java, können wir auch **anonyme Funktionen** in funktionalen Sprachen definieren
- Werden **Lambda**-Funktionen genannt, da sie in der Theorie mit dem λ -Zeichen deklariert werden
- Beispiel:

$$f(g(x,y), x, y) = g(x,y) / x$$

$$\begin{aligned} f(\lambda(x,y)=x+y, 2, 4) &= 3 \\ f(\lambda(x,y)=x*y, 2, 4) &= 4 \end{aligned}$$

Funktionale Programmierung in Java

JAVA 8 LAMBDAS

ACHTUNG: Generics!

- Beispiel: Wir wollen Personen filtern.
 - Gib alle Personen aus, die die Filter-Bedingung erfüllen.
- Filter wird mit Interface definiert:

```
public interface Tester<T> {  
    boolean test(T t);  
}
```

- Ausgabe mit Filter:

```
public void printPersons(Tester<Person> tester, List<Person> roster) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```

- Konkreter Filter:
 - Für den Typ Person
 - Erlaubt männliche Personen in bestimmter Altersgruppe

```
public class SomeSpecificTester implements Tester<Person> {  
    @Override  
    public boolean test(Person t) {  
        return (t.getAge() >= 18 && t.getAge() <=25 && p.getGender() == person.Sex.MALE);  
    }  
}
```

- Anwendung des Filters:

```
printPersons(new SomeSpecificTester(), roster);
```

- **Ergebnis:** Wir können Funktionen übergeben, in dem wir Interfaces für diese spezifizieren und konkrete Implementierungen übergeben.

- Wir können uns die Definition der separaten Klassen, welche von Interface `Tester<T>` erben vereinfachen.
- Anonyme Klasse beim Aufruf angeben:

```
printPersons (new Tester<Person>() {  
    @Override  
    public boolean test(Person t) {  
        return (t.getAge() >= 18 && t.getAge() <=25 && p.getGender() == person.Sex.MALE);  
    }  
}, roster);
```

- So müssen wir nur noch den Filter direkt im Aufruf definieren und können uns die Definition einer weiteren Klasse sparen.

- `Tester<T>` ist ein **funktionales Interface**.
 - Ein funktionales Interface darf nur eine Methode enthalten.
 - Java erkennt diese einzige Methode und typisiert den Lambda-Ausdruck automatisch.

```
printPersons(  
    (Person p) -> p.getGender() == Person.Sex.MALE  
                && p.getAge() >= 18  
                && p.getAge() <= 25,  
    roster  
);
```

} **Lambda-Ausdruck**

- Syntax:
(Eingabetyp Bezeichner) -> Ausdruck mit Rückgabetyp

Methode

Parametertypen: (*Tester<Person>*, *List<Person>*),
Tester<Person> ist ein **funktionales Interface**.

```
public interface Tester<T> {  
    boolean test(T t);  
}
```

Funktionales Interface

Tester<T> hat nur eine Methode mit Rückgabotyp **boolean** und generischen Parameter **T**

Eingabeparameter

p vom Typ **Person** als Eingabeparameter von *test(T t)*. Der anonyme Tester wird also als *Tester<Person>* instanziiert.
T wird zu **Person**.

```
printPersons(  
    (Person p) ->  
        p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25,  
    roster  
);
```

Pfeil-Operator

-> trennt Eingabe-Parameter vom Funktionalen Ausdruck.

Ausdruck

Muss den Rückgabewert der Funktion *test(Person t)* haben
→ **boolean**

- *java.util.function* bietet generische funktionale Interfaces
- Für unseren Fall:

```
public interface Predicate<T> {  
    boolean test(T t);  
}
```
- Erspart die Definition von eigenen Interfaces!
- Neue `printPersons`-Methode:

```
public void printPersonsWithPredicate(List<Person> roster, Predicate<Person> tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```


- Neue printPersons-Methode:

```
public void printPersonsWithPredicate(List<Person> roster, Predicate<Person> tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```

- Aufruf:

```
printPersonsWithPredicate(  
    roster,  
    p -> p.getGender() == Person.Sex.MALE  
        && p.getAge() >= 18  
        && p.getAge() <= 25  
);
```

Kann auch
automatisch
typisiert
werden.



- Wir wollen die auszuführende Aktion bestimmbar machen
 - Nicht immer nur `p.printPerson()` ;
- `Consumer<T>`: funktionales Interface

```
public interface Consumer<T> {  
    void accept(T t);  
}
```

```
public void processPersons(List<Person> roster, Predicate<Person> tester,  
    Consumer<Person> block) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            block.accept(p);  
        }  
    }  
}
```

- Aufruf:

```
processPersons(roster,  
    p -> p.getGender() == Person.Sex.MALE  
    && p.getAge() >= 18  
    && p.getAge() <= 25,  
    p -> p.printPerson()  
);
```

- Wir wollen die Email-Adresse der Person ermitteln und dann ausgeben.

- `Function<T, R>`

```
public interface Function<T,R> {  
    R apply(T t);  
}
```

```
public void processPersonsWithFunction(List<Person> roster, Predicate<Person> tester,  
    Function<Person, String> mapper, Consumer<String> block) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            String data = mapper.apply(p);  
            block.accept(data);  
        }  
    }  
}
```

- Aufruf:

```
processPersonsWithFunction(roster,  
    p -> p.getGender() == Person.Sex.MALE  
    && p.getAge() >= 18  
    && p.getAge() <= 25,  
    p -> p.getEmailAddress(),  
    email -> System.out.println(email)  
);
```

email wird
automatisch mit dem
Ergebnis des
Mappers befüllt.

- `Iterable<X>` ist das Interface, was von allen Klassen mit Iteratoren implementiert wird.
 - Arrays
 - Collections
- Komplett generische Version:

```
public void processElements (Iterable<X> source, Predicate<X> tester,
    Function<X, Y> mapper, Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

- Aufruf bleibt gleich.

- Werden auf Streams ausgeführt.
 - Sind eine Objektsammlung
 - Erhält man mit der `stream()`-Methode.

- Beispiel:

```
roster
    .stream()
    .filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```

- Operationen:

Aktion	Operation
Filtere Objekte, die das Predicate erfüllen	<code>Stream<T> filter(Predicate<? super T> predicate)</code>
Bilde Objekte mit einem Function-Objekt auf einen anderen Wert ab	<code><R> Stream<R> map(Function<? super T,? extends R> mapper)</code>
Führe eine Aktion mit einem Consumer-Objekt aus	<code>void forEach(Consumer<? super T> action)</code>

Fragen?

