13. Lambdas

- In diesem Kapitel geht es um wichtige Konzepte in Java, um den Programmieraufwand bei typischen Problemen zu reduzieren. Dies bedeutet:
  - wenige Zeilen von Code
  - wenige Fehler

    ➔ Niedrigere Kosten bei der Softwareerstellung
  - Folgende Konzepte werden vorgestellt:
    - Funktionale Programmierung in Java (Lambdas)
    - Streams

Motivation

- Problem
  - Verwaltung einer Menge von Objekten, wie z. B. POIs (Poinst of Interests) oder Musiker, in einer Liste.
  - Unterstützung von Suchoperationen, auch als **Filter** bezeichnet.
    - Welche POI (Points Of Interests) liegen in der Umgebung von Marburg?
    - Zeige mir die Musiker mit Namen "Wilson"?
  - Jede Suchoperation soll die Ergebnisse elementweise als Iterator liefern.
- Erste Lösung
  - Die Elemente aus der Liste<Musician> können die Eingabe über die Methode iterator() als Iterator geliefert werden.
  - Für jede Anfrage wird eine eigene Iterator-Klasse implementiert.

Lösung mit einem Iterator

```
class FilterMusician implements Iterator<Musician> {
       Iterator<Musician> it;     // Iterator, der über alle Element geht.
       Musician cur = null;       // Musician, der aktuell gefunden wurde.

       public FilterMusician(Iterator<Musician> in) {
          it = in;                  // Speichern des Iterators
          cur = computeNext();      // Nächstes Ergebnis berechnen
        }

        private Musician computeNext() {
          while (it.hasNext()) {                  // Durchlauf bis zum nächsten Ergebnis
             Musician tmp = it.next();
             if (tmp.getName().equals("Wilson")) // Prüfen der Bedingung
               return tmp;                        // Treffer
          }
          return null;                            // Kein Ergebnis
        }

        public boolean hasNext() {
          return (cur != null);
        }

        public Musician next() {
           Musician tmp = cur;
           cur = computeNext();   // Nächstes Ergebnis berechnen
           return tmp;
        }
}
```

Lösung mit einem Iterator

```
class FilterMusician implements Iterator<Musician> {
      Iterator<Musician> it;     // Iterator, der über alle Element geht.
      Musician cur = null;       // Musician, der aktuell gefunden wurde.

      public FilterMusician(Iterator<Musician> in) {
         it = in;                  // Speichern des Iterators
         cur = computeNext();      // Nächstes Ergebnis berechnen
       }

       private Musician computeNext() {
         while (it.hasNext()) {                  // Durchla   bis zum nächsten Ergebnis
            Musician tmp = it.next();
            if (tmp.getName().equals("Wilson")) // Prüfen der
              return tmp;                       // Treffe
         }
         return null;                           // Kein E
       }

      public
        retu
      }

      public
         Mus
         cur                                    bnis berechnen
         ret
      }
}
```

- aufwändig
- redundant
- fehleranfällig
- viel Programmcode für wenig Funktionalität

Für alle Filterbedingung müssten eine eigene Implementierungen angefertigt werden, die sich nur in wenigen Punkten unterscheiden.

Besser: Test in eigene Klasse auslagern

```
class FilterMusician implements Iterator<Musician> {
       Iterator<Musician> it;     // Iterator, der über alles geht.
       MusicianPredicate condition;  // Filterbedingung
       Musician cur = null;       // Musician, die aktuell gefunden wurde.

       public FilterMusician(Iterator<Musician> in,
                             MusicianPredicate c) {
         it = in;                   // Speichern des Iterators
         condition = c;             // Speichern der Bedingung
         cur = computeNext();    // Nächstes Ergebnis berechnen
       }

       private Musician computeNext() {
         while (it.hasNext()) {          // Durchlauf bis zum nächsten Ergebnis
           Musician tmp = it.next();
           if (condition.test(tmp))    // Prüfen der Bedingung
             return tmp;                   // Treffer
         }
         return null;                     // Kein Ergebnis
       }

       ...
}
```

Noch besser: Generics benutzen

```
class FilterIterator<T> implements Iterator<T> {
       Iterator<T> it;                  // Iterator, der über alles geht.
       Predicate<T> condition;  // Filterbedingung
       T cur = null;                     // Element, die aktuell gefunden wurde.

       public FilterIterator(Iterator<T> in, FilterCondition<T> c) {
         it = in;                 // Speichern des Iterators
         condition = c;           // Speichern der Bedingung
         cur = computeNext();  // Nächstes Ergebnis berechnen
       }

       private T computeNext() {
         while (it.hasNext()) {          // Durchlauf bis zum nächsten Ergebnis
           T tmp = it.next();
           if (condition.test(tmp))   // Prüfen der Bedingung
             return tmp;                   // Treffer
         }
         return null;                     // Kein Ergebnis
       }

       ...

}
```

Diskussion

- Vorteile
  - Wiederverwendung der Implementierung des FilterIterator
  - Minimale Implementierung der Filterbedingung

  ➔ Keine Duplizierung des Codes

```
public class WilsonFilter implements Predicate<Musician> {
        boolean test( Musician p ) {
                return p.getName().equals("Wilson");
        }
}
```

- Nachteile
  - Bei vielen Filterbedingungen gibt es auch viele Klassen
  - Diese werden meistens nur an einer Stelle genutzt

  ➔ Benennung der Klasse scheint überflüssig

13.1 Lambda-Ausdrücke in Java 8

- Motivation
  - Wir werden im Folgenden sehen, dass durch das in Java 8 neue Konzept von Lambda-Ausdrücken (kurz Lambdas) sich Filter sehr einfach hinschreiben lassen.
- Grundlage hierfür sind sogenannte funktionale Schnittstellen (engl.: Functional Interfaces) in Java.
  - Interface mit nur einer abstrakten Methode (mit Ausnahme der Methoden, die es in der Klasse Object noch gibt).
  - Da bei diesen Interfaces nur eine Methode existiert, kann man den Datentyp des Innterfaces als Datentyp dieser Methode betrachten.
    - Zuweisung an Variablen
    - Übergabe als Parameter
    - Rückgabewert von Methoden

Vorhandene funktionale Interfaces

- In dem Paket **java.util.function** werden die in Java vorhandenen funktionale Interfaces bereitgestellt.
  - Consumer<T> mit der Methode void accept(T)
  - Predicate<T> mit der Methode boolean test(T)
    - Damit lassen sich sehr gut Filterbedingungen ausdrücken, um aus Mengen von Objekten einen Teil zu extrahieren.
  - Function<T,R> mit der Methode R apply(T)
    - Damit lassen sich beliebige Transformationen von dem Typ T auf den Typ R ausdrücken.
  - Supplier<T> mit der Methode T get()
    - Damit kann die Erzeugung von Objekten abgebildet werden.

  …

Beispiel: Funktionales Interface Consumer

- Implementierung des Codes, der für jedes Listenelement ausgeführt werden soll als **Lambda**
- Die Methode (der funktionalen Schnittstelle) wird ohne umgebende Klasse implementiert:

```
public static void main(String[] args) {
        List<Musician> mlist = new LinkedList<>();
        …
        mlist.forEach(
        p -> {
                if (p.getName().equals("Wilson"))
                        System.out.println(p);
             } );
}
```

Beispiel: Funktionales Interface Consumer

- Implementierung des Codes, der für jedes Listenelement ausgeführt werden soll als **Lambda**
- Die Methode (der funktion
  umgebende Klasse imple

Noch nicht exakt dasselbe wie der Filter, da hier die Filterung (Bedingung) und Ausgabe der Elemente (Ausgabe) zusammen implementiert ist. Eine bessere Lösung sehen wir noch später.

```
public static void mai
        List<Musician>
        …
        mlist.forEach(
        p -> {
                if (p.getName().equals("Wilson"))
                        System.out.println(p);
            } );
}
```


Kurzer Rückblick

- Das sogenannte Lambda-Kalkül wurde 1930 von Alonzo Church vorgestellt.
  - Mathematiker wie **Alonzo Church** und **Alan Turing** haben grundlegende **Modelle zur Berechenbarkeit** entwickelt.
    - Sie zeigten, dass **nicht alle Funktionen berechenbar** sind.
    - Turing-Berechenbarkeit und das Lambda-Kalkül sind gleichmächtig.
- Das Lambda-Kalkül ist die Grundlage von **funktionalen Programmiersprachen**, wie z. B. Lisp und Haskell.
  - siehe auch das Modul Deklarative Programmierung
- Inzwischen wird auch in vielen objektorientierten Sprachen, wie z. B. Java, C++, C# und Scala, das Konzept des Lambda-Kalküls unterstützt.

Was ist ein Lambda?

- Ein Lambda ist eine anonyme Funktion.
  - Eine anonyme Funktion ist eine Funktion ohne Namen.
  - Die typische Schreibweise besteht aus
    (Liste der Parameter) -> {Funktionsrumpf} wie z. B.
    - `(int x, int y) -> {return x + y;} // Addition`
  - In gewissen Situation darf man noch Dinge bei der Angabe eines Lambdas weglassen. Z. B.:
    - `x -> {return x * x;}  // Parameter ohne Typ`
    - `( ) -> x                    // Funktionsrumpf mit einer`
      `// Codezeile.`

Besonderheiten bei Lambdas

- Ein Lambda hat keinen, einen oder mehrere durch Komma getrennte Parameter.
  - Der Typ der Parameter muss nur dann angegeben werden, wenn dieser sich aus dem Kontext nicht eindeutig ergibt.
  - Bei einem Parameter (ohne Angabe des Typs) werden Klammern nicht benötigt.
  - ( ) bezeichnet eine Funktion ohne Parameter
- Der Methodenrumpf besteht aus einer Folge von Anweisungen.
  - Bei nur einer Anweisung (ohne Kontrollstrukturen) können die Mengenklammern und das Semikolon weggelassen werden.

Weitere Vorgehensweise

- Exemplarisch wird die Syntax der Lambdas in Java 8 vorgestellt.
- Funktionstypen
- Methodenreferenzen
- Datenströme (Streams)

Beispiel 1: Ausgabe einer Liste

```
public class LambdaTest {

    public static void main(String[] args) {
        LinkedList<Integer> values = new LinkedList<Integer>();
        values.addAll(Arrays.asList(42,33,7,88,23));
        values.forEach(x -> System.out.println(x));
    }
}
```

- `x -> System.out.println(x)` ist ein Lambda mit einem Parameter und einer Anweisung.
  - Der Typ des Parameters ergibt sich aus dem Kontext. Da values eine Liste mit Integer-Objekten ist und forEach den Lambda für jedes Element aufruft, muss x vom Typ Integer sein.
- Bei einzeiligen Lambdas mit Rückgabetyp kann auch auf die Angabe des Schlüsselwort return verzichtet werden.
  - Das Ergebnis des Ausdrucks ist gleich dem Ergebnis der Methode.

Beispiel 2: Rumpf mit mehreren Anweisungen

```
public class LambdaTest {

    public static void main(String[] args) {
        LinkedList<Integer> values = new LinkedList<Integer>();
        values.addAll(Arrays.asList(42,33,7,88,23));
        values.forEach(x -> {
                                x += 7;
                                System.out.println(x);
                            });
    }
}
```

- Hierbei werden die Klammern benötigt, um aus mehreren Anweisungen einen Block zu erzeugen.

Beispiel 3: Lambda mit lokalen Variablen

```
public class LambdaTest {

    public static void main(String[] args) {
        LinkedList<Integer> values = new LinkedList<Integer>();
        values.addAll(Arrays.asList(42,33,7,88,23));
        values.forEach(x -> {
                                int y = x+7;
                                System.out.println(y);
                            });
    }
}
```

- Man kann wie üblich lokale Variablen im Rumpf eines Lambdas deklarieren.

Beispiel 4: Lambda mit expliziter Typangabe

```
public class LambdaTest {

    public static void main(String[] args) {
        LinkedList<Integer> values = new LinkedList<Integer>();
        values.addAll(Arrays.asList(42,33,7,88,23));
        values.forEach((Integer x) -> {
                                        int y = x+5;
                                        System.out.println(y);
                                      });
    }
}
```

- Bei Bedarf kann man auch explizit den Typ zu dem Parameter hinzufügen.
  - Hierzu muss aber der Parameter geklammert werden.

Lambda und Funktionstypen

- Lambdas besitzen einen **Typ**, der über eine funktionale Schnittstelle beschrieben ist.
  - z. B. Consumer<T> mit der abstrakten Methode void accept(T).
- Ein Lambda mit einem Parameter kann man z. B. an eine **Variable** vom Typ Consumer zuweisen.
  - Die Variable repräsentiert dann diese Funktion.
  - Damit kann diese Funktion als Wert z.B. an eine Methode übergeben werden.
  - Die Funktion kann über den Methodennamen der funktionalen Schnittstelle angesprochen werden.

```
public class LambdaTest {

    public static void main(String[] args) {
        LinkedList<Integer> values = new LinkedList<Integer>();
        values.addAll(Arrays.asList(42,33,7,88,23));
        Consumer ic = x -> System.out.println(x);
        values.forEach(ic);    }
}
```

Lambda als Rückgabewert

- Man kann jetzt auch **Lambdas** in einer Methode erzeugen und diese dann **als Rückgabewert** liefern.
  - Der Typ des Rückgabewerts muss einer passenden funktionalen Schnittstelle entsprechen.

```
static Consumer<Integer> getConsumer() {
    return x -> System.out.println(x);
}

public static void main(String[] args) {
     LinkedList<Integer> values = new LinkedList<Integer>();
     values.addAll(Arrays.asList(42,33,7,88,23));
     values.forEach(getConsumer());
}
```

13.2 Methodenreferenzen

- **Motivation**
  - Eine vorhandene Methode (und ein Konstruktor) sollte statt eines Lambdas genutzt werden.
    - Ein Umweg über Lambdas sollte hier vermieden werden, wenn die Funktionalität bereits vorhanden ist.
  - Jedoch muss die Methode bzw. der Konstruktor zu der abstrakten Methode des Lambdas (funktionalen Interface) passen.

- **Definition**
  - Eine Methodenreferenz ist ein Verweis auf eine Methode.
  - Syntax
    - Methodenreferenzen verwenden den Methodennamen (bzw. bei Konstruktoren das Schlüsselwort new).
    - Davor steht noch :: und der Klassenname bzw. die Objektreferenz.

Beispiel

- Die Signatur von PrintStream.println(Object) ist kompatibel zu Consumer<T>.accept(T):

```
public class MusicianPrinter implements Consumer<Musician> {
    public static void main(String[] args) {
            List<Musician> ps = Arrays.asList(...);
            ps.forEach( System.out::println );
    }
}
```

  - Gibt alle Elemente der Liste auf der Konsole aus.

Methodenreferenzen (im Detail)

- Folgende vier Fälle werden bei Methodenreferenzen unterschieden:

Fall	Art der Methode	Syntax	Beispiel
1	Statische Methode	ClassName::StaticMethodName	String::valueOf
2	Konstruktor	ClassName::new	ArrayList::new
3	Objektmethode via Objekt	objectReference::MethodName	x::toString
4	Objektmethode via Klasse	ClassName::MethodName	Object::toString

Ziel-Objekt wird dann als erstes Argument übergeben.

Statische Methodenreferenzen

- Syntax der Referenzangabe
  - ClassName::StaticMethodName
- Beispiel

```
class A {
    static void printInt(int x){
        System.out.println(x);
    }
}

public class TestMethodReferences {
    public static void main(String[] args) {
        LinkedList<Integer> values = new LinkedList<>();
        values.addAll(Arrays.asList(1,2,3,4,5));
        // Fall 1:
        Consumer<Integer> c = A::printInt; // c: Integer --> ()
        values.forEach(c);
    }
}
```

Konstruktoren

- Syntax der Referenzangabe
  - ClassName::new
- Beispiel

```
class A {
    private int i;
    public A(int x) { i = x; }
    public String toString() { return "Klasse A: " + i; }
}

public class TestMethodReferences {
    public static void main(String[] args) {
        LinkedList<Integer> values = new LinkedList<>();
        values.addAll(Arrays.asList(1,2,3,4,5));
        // Fall 2:
        Function<Integer, A> f = A::new; // f: Integer --> A
        System.out.println("Ausgabe " + f.apply(42));
    }
}
```

Objektmethoden via Objekt

- Syntax der Referenzangabe
  - ObjectReference::MethodName
- Beispiel

```
class A {
    private int i;
    public A(int x) { i = x; }
    public String toString() { return "Klasse A: " + i; }
}

public class TestMethodReferences {
    public static void main(String[] args) {
        LinkedList<Integer> values = new LinkedList<>();
        values.addAll(Arrays.asList(1,2,3,4,5));
        // Fall 3:
        A a = new A(7);
        Supplier<String> s = a::toString;   // () --> String
        System.out.println("Ausgabe " + s.get());
    }
}
```

Objektmethode via Klasse

- Syntax der Referenzangabe
  - ClassName::MethodName
- Beispiel

```
class A {
    private int i;
    public A(int x) { i = x; }
    public String toString() { return "Klasse A: " + i; }
}

public class TestMethodReferences {
    public static void main(String[] args) {
        LinkedList<Integer> values = new LinkedList<>();
        values.addAll(Arrays.asList(1,2,3,4,5));
       A a = new A(8);
        // Fall 4:
        Function<A, String> g = A::toString;  // A --> String
        System.out.println("Ausgabe " + g.apply(a));
    }
}
```