7 Anonyme Klassen und Lambda-Ausdrücke

7.1 Anonyme Klassen

- Wenn von einer Klasse nur ein einziges Objekt benötigt wird, kann man auf eine explizite Deklaration der Klasse verzichten und stattdessen eine *anonyme* Klasse verwenden.

- Eine anonyme Klasse wird definiert, indem dem Ausdruck `new type (arguments)` zur Erzeugung ihres einzigen Objekts ein Klassenrumpf `{ ..... }` nachgestellt wird.

- Wenn `type` eine Klasse bezeichnet, dann ist die anonyme Klasse eine direkte Unterklasse dieser Klasse.
  Wenn `type` eine Schnittstelle bezeichnet, dann ist die anonyme Klasse eine direkte Unterklasse der Wurzelklasse `Object`, die diese Schnittstelle implementiert.

- Eine anonyme Klasse kann keinen expliziten Konstruktor besitzen.
  Sie besitzt implizit einen öffentlichen Konstruktor, der die Aufrufparameter `arguments` unverändert an den entsprechenden Oberklassenkonstruktor weiterreicht.

- Anonyme Klassen unterliegen gewissen Einschränkungen; beispielsweise dürfen sie keine statischen Elemente enthalten.

Beispiel

- Ausgabe einer Wertetabelle für eine beliebige Funktion, die als Parameter übergeben wird

Lösung mit Funktionszeigern in C

- Siehe § 2.2.6.

Lösung in Java

```
// Schnittstelle (oder auch abstrakte Klasse) zur Repräsentation
// von Funktionen mit Parameter- und Resultattyp double.
interface Function {
  // Funktionswert an der Stelle x berechnen.
  double compute (double x);
}

class Table {
  // Wertetabelle der Funktion f für x von x1 bis x2
  // mit Schrittweite dx ausgeben.
  public static
  void print (Function f, double x1, double x2, double dx) {
    for (double x = x1; x <= x2; x += dx) {
      System.out.println(x + "\t" + f.compute(x));
    }
  }
}
```

```
// Explizite Klasse zur Repräsentation der Funktion x*x.
class F1 implements Function {
  // Geeignete Implementierung der Schnittstellen-Methode compute.
  public double compute (double x) { return x*x; }
}

class Test {
  public static void main (String [] args) {
    // Wertetabelle von x*x ausgeben.
    Table.print(new F1(), 1, 10, 1);

    // Wertetabelle von 1/x ausgeben.
    Table.print(
      // Objekt einer anonymen Klasse erzeugen,
      // die die Schnittstelle Function implementiert
      // und hierfür eine geeignete Implementierung
      // der Methode compute enthält.
      new Function () {
        public double compute (double x) { return 1/x; }
      },
      1, 10, 1);
  }
}
```

7.2 Lambda-Ausdrücke

- Ein *Lambda-Ausdruck* definiert logisch eine anonyme Funktion und besteht aus einer Parameterliste, einem Pfeil (->) und einer Implementierung.

- Die Parameterliste ist
  - entweder genauso aufgebaut wie die Parameterliste einer Methode
  - oder eine geklammerte Liste von Namen ohne Typen, durch Kommas getrennt,
  - oder ein einzelner Name ohne Typ und ohne Klammern.

- Die Implementierung ist
  - entweder ein Anweisungsblock wie bei einer Methode
  - oder ein Ausdruck.

- Eine *funktionale Schnittstelle* (functional interface) ist eine Schnittstelle, die (neben eventuellen anderen Elementen) genau eine abstrakte Methode enthält (z. B. die Schnittstelle `Function` aus § 7.1).

- Ein Lambda-Ausdruck muss immer auf eine der folgenden Arten im *Kontext* einer funktionalen Schnittstelle `F` verwendet werden:

- Er wird an eine Variable mit Typ `F` zugewiesen.
- Er wird an einen Parameter mit Typ `F` übergeben.
- Er wird durch eine Typumwandlung (cast) in den Typ `F` umgewandelt.

- Technisch liefert ein Lambda-Ausdruck ein Objekt einer anonymen Klasse, die die funktionale Schnittstelle `F` implementiert, indem sie folgende Methode definiert:
  - Der Name und der Resultattyp werden von der Methode `m` der Schnittstelle `F` übernommen.
  - Die Parameter werden vom Lambda-Ausdruck übernommen.

    Wenn sie keine Typen besitzen, werden sie von den Parametern der Methode `m` übernommen.
    Andernfalls müssen sie mit den Typen dieser Parameter übereinstimmen.

    In jedem Fall muss die Anzahl der Parameter übereinstimmen.
  - Die Implementierung wird vom Lambda-Ausdruck übernommen.

    Wenn es sich um einen Anweisungsblock handelt, müssen alle darin enthaltenen `return`-Anweisungen zum Resultattyp der Methode passen.

    Wenn es sich um einen Ausdruck `expr` handelt, wird er in einen Anweisungsblock `{ expr; }` (wenn der Resultattyp der Methode `void` ist) oder `{ return expr; }` (andernfalls) umgewandelt.

Beispiel (vgl. § 7.1)

```
class TestLambda {
  public static void main (String [] args) {
    // Wertetabelle von x*x ausgeben.
    // Der Lambda-Ausdruck kann unterschiedlich formuliert werden.

    Table.print((double x) -> { return x * x; }, 1, 10, 1);
    Table.print((x) -> { return x * x; }, 1, 10, 1);
    Table.print(x -> { return x * x; }, 1, 10, 1);

    Table.print((double x) -> x * x, 1, 10, 1);
    Table.print((x) -> x * x, 1, 10, 1);
    Table.print(x -> x * x, 1, 10, 1);
  }
}
```

Vordefinierte funktionale Schnittstellen

- Das Paket `java.util.function` enthält zahlreiche vordefinierte funktionale Schnittstellen, zum Beispiel `DoubleUnaryOperator` mit einer Methode

  ```
  double applyAsDouble (double x);
  ```

  oder `IntBinaryOperator` mit einer Methode

  ```
  int applyAsInt (int x, int y);
  ```

- Deshalb könnte die Methode `print` der Klasse `Table` in § 7.1 anstelle der selbstdefinierten Schnittstelle `Function` auch die vordefinierte Schnittstelle `java.util.function.DoubleUnaryOperator` verwenden, wenn `f.compute` durch `f.applyAsDouble` ersetzt wird.

7.3 Verwendung lokaler Variablen

- Lokale Variablen und Parameter aus der „Umgebung“ einer anonymen Klasse oder eines Lambda-Ausdrucks dürfen dort nur verwendet werden, wenn sie unveränderlich oder faktisch unveränderlich sind (vgl. § 4.11). Insbesondere dürfen sie dort nicht verändert werden.
  (Außerdem müssen sie garantiert vorher einen Wert erhalten haben.)

- Hintergrund:
  - Das Objekt, das aus einer anonymen Klasse oder einem Lambda-Ausdruck entsteht, kann eine längere Lebensdauer besitzen als die genannten lokalen Variablen.
  - Wenn sie im Code der Klasse bzw. des Lambda-Ausdrucks uneingeschränkt verwendet werden dürften, könnte bei der Ausführung dieses Codes auf nicht mehr vorhandene Variablen zugegriffen werden.
  - Wenn die Variablen (faktisch) unveränderlich sind, können sie vom Übersetzer im Code der Klasse bzw. des Lambda-Ausdrucks direkt durch ihre aktuellen Werte ersetzt werden, sodass bei der Ausführung dieses Codes nicht mehr auf sie zugegriffen werden muss.

- Die genannte Einschränkung kann wie folgt umgangen werden:
  - Wenn in einer anonymen Klasse oder einem Lambda-Ausdruck nur lesend auf eine lokale Variable aus der Umgebung zugegriffen wird, kann vorher eine (faktisch) unveränderliche Kopie von ihr erstellt werden.

    Zum Beispiel:

```
// Wertetabellen der Funktionen 1*x bis 5*x ausgeben.
for (int m = 1; m <= 5; m++) {
  // Die Laufvariable m darf im nachfolgenden Lambda-
  // Ausdruck nicht verwendet werden, weil sie nicht
  // (faktisch) unveränderlich ist.
  // Die Hilfsvariable mm (die logisch in jedem Schleifen-
  // durchlauf neu angelegt wird) ist jedoch faktisch
  // unveränderlich und darf deshalb verwendet werden.
  int mm = m;
  Table.print(x -> mm * x, 1, 10, 1);
}
```

  - Andernfalls kann anstelle einer Variablen das Element einer einelementigen Reihe verwendet werden, das problemlos verändert werden kann, auch wenn die Variable, in der die Reihe gespeichert ist, (faktisch) unveränderlich ist.

Beispiel (vgl. § 4.13 und § 5.14)

```
// Funktionale Schnittstelle für Aktionen,
// die mit einem int-Wert als Parameter aufgerufen werden.
interface Action {
  void execute (int x);
}

// Liste von int-Werten.
class List {
  protected int head;   // Erstes Element (Kopf).
  protected List tail;  // Restliste (Schwanz).

  // Initialisierung mit erstem Element h und ggf. Restliste t.
  public List (int h, List t) { head = h; tail = t; }
  public List (int h) { this(h, null); }

  // Erstes Element und Restliste abfragen.
  public int head () { return head; }
  public List tail () { return tail; }
```

```
  // Aktion a für jedes Element der Liste aufrufen.
  public void forEach (Action a) {
    for (List p = this; p != null; p = p.tail) {
      a.execute(p.head);
    }
  }

  // Liste ausgeben.
  public void print () {
    forEach(x -> System.out.println(x));
  }

  // Länge (d. h. Anzahl der Elemente) der Liste ermitteln.
  public int length () {
    // Einelementige int-Reihe anstelle einer int-Variablen,
    // weil eine Variable im Lambda-Ausdruck nicht verändert
    // werden darf.
    int [] n = { 0 };
    forEach(x -> n[0]++);
    return n[0];
  }
}
```

```
// Zirkuläre Liste.
class CircList extends List {
  // Initialisierung mit erstem Element h und ggf. Restliste t.
  public CircList (int h, CircList t) {
    super(h, t);

    // Zirkuläre Verkettung herstellen.
    if (t == null) {
      tail = this;
    }
    else {
      List p = t;
      while (p.tail != t) p = p.tail;
      p.tail = this;
    }
  }
  public CircList (int h) { this(h, null); }

  // Überschreibung von tail mit kovarianter Anpassung
  // des Resultattyps (CircList statt List).
  public CircList tail () { return (CircList)tail; }
```

```
    // Anders als in § 5.14 muss hier lediglich forEach überschrieben
    // werden, um die Zirkularität zu beachten.
    // print und length (und eventuelle weitere Methoden, die forEach
    // verwenden, können unverändert bleiben.)
    public void forEach (Action a) {
      a.execute(head);
      for (List p = tail; p != this; p = p.tail) {
        a.execute(p.head);
      }
    }
  }
```