4.5 Reihungen und Zeichenketten

4.5.1 Reihungen (Arrays)

- Beipiel: Algorithmus *Wechselgeld 3*

  Führe folgende Schritte der Reihe nach aus:

  1. Berechne *d = 100 – r und setze w = ().*
  2. Solange *d ≥ 5: Vermindere d um 5 und nimm 5 zu w hinzu.*
  3. Falls *d ≥ 2, dann vermindere d um 2 und nimm 2 zu w hinzu.*
  4. Falls *d ≥ 2, dann vermindere d um 2 und nimm 2 zu w hinzu.*
  5. Falls *d ≥ 1, dann vermindere d um 1 und nimm 1 zu w hinzu.*

  - Umsetzung in Java?
    - Ausgabe: Array in denen entsprechend viele 5er, 2er und 1er stehen
    - Problem: wir wissen vorher nicht genau, wie viele Scheine/Münzen wir zurück geben, d.h. wie lang das Array sein wird
    - Lösung: wir merken uns, wie viele 5er, 2er und 1er wir jeweils ausgeben

```
public static int[] wechselgeld(int betrag) {
    int fuenfer = 0;          // Wieviel 5er
    int zweier = 0;           // Wieviel 2er
    int einser = 0;           // Wieviel 1er
    int diff = 100 - betrag;

    while(diff >= 5) { diff = diff - 5; fuenfer++; }

    while(diff >= 2) { diff = diff - 2; zweier++; }

    if(diff == 1) { einser++; }

    // Ergebnis-Array erzeugen
    int[] erg = new int[fuenfer+zweier+einser];
    for(int i=0; i<fuenfer; i++) {
          erg[i] = 5;
    }
    for(int i=fuenfer; i<fuenfer+zweier; i++) {
          erg[i] = 2;
    }
    for(int i=fuenfer+zweier; i<fuenfer+zweier+einser; i++) {
          erg[i] = 1;
    }
    return erg;
}
```

4.5 Reihungen und Zeichenketten

4.5.1 Reihungen (Arrays)

- Da auch Arrays einen bestimmten Typ haben z.B. `gruss : char[]` kann man auch Reihungen von Reihungen bilden. Mit einem Array von Arrays lassen sich z.B. Matrizen modellieren:

```
int[] m0 = {1, 2, 3};
int[] m1 = {4, 5, 6};
int[][] m = {m0, m1};
```

- Daher heißen diese Gebilde auch *mehrdimensionale* Arrays.

- Offensichtlich sind Arrays über dem Typ *CHAR* bzw. **char** *Zeichenketten* (*Strings*)
- Java stellt einen eigenen Typ `String` für Zeichenketten zur Verfügung, d.h. es gibt eine eigene Sorte (mit Operationen) für Zeichenketten in Java, wir können mit diesem Typ ganz normal „arbeiten“

```
public class HelloWorld
{
  public static final String GRUSS = "Hello, World!";

  public static void main(String[] args)
  {
    System.out.println(GRUSS);
  }
}
```

- In der Deklaration und Initialisierung

  `public static` `String gruss = "Hello, World!";`

  entspricht der Ausdruck "Hello, World!" einer speziellen Schreibweise für ein konstantes Array **`char`**`[13]`, das in einen Typ `String` „gekapselt” wird.
- **Achtung**: Die Komponenten dieses Arrays können nicht mehr (durch Neuzuweisung) geändert werden.

4.5 Reihungen und Zeichenketten

4.5.2 Zeichenketten (Strings)

- Der Typ **`String`** ist kein *primitiver* Typ, sondern eine Klasse von Objekten, ein sog. *Objekttyp*.
- Werte dieses Typs können aber – wie bei primitiven Typen – durch Literale gebildet werden (in “” eingeschlossen).
- Beispiele für Literate der Sorte `String` in Java:
  - `"Hello, World!"`
  - `"Kroeger"`
  - `"Guten Morgen"`
- Literale und komplexere Ausdrücke vom Typ `String` können durch den (abermals überladenen!) Operator + konkateniert werden.
  - `"Guten Morgen"+" "+"Kroeger"` ergibt die Zeichenkette `"Guten Morgen Kroeger"`

Die Klasse String bietet verschiedene Operationen (dahinter stehen statische Methoden)

- Typcast, um aus primitiven Typen Strings zu erzeugen:

```
static String valueOf(boolean b)
static String valueOf(char c)
static String valueOf(char[] c)
static String valueOf(int i)
static String valueOf(long l)
static String valueOf(float f)
static String valueOf(double d)
```

Bei der Konkatenation (z.B. `"Note: "+1.0`) werden diese Methoden (hier: `static String valueOf(double d)`) implizit verwendet.

- Da es sich intern um ein Array handelt, gibt es auch die beiden Operationen
  - Länge der Zeichenkette durch die Methode `int length()`.
  - Die Methode `char charAt(int index)` liefert das Zeichen an der gegebenen Stelle des Strings.
    - Das erste Element hat den Index `0`.
    - Das letzte Element hat den Index `length() - 1`.
  - Beispiel:
    - der Ausdruck `"Hello, World!".length` hat den Wert: 13
    - der Ausdruck `"Hello, World!".charAt(12)` hat den Wert: `'!'`

Hinweis:

Es gibt noch viele weitere Operationen auf dem Typ `String`, die wir uns später nochmal anschauen.

4.5 Reihungen und Zeichenketten

4.5.2 Zeichenketten (Strings)

- Nun verstehen Sie auch endlich offiziell den Parameter der `main`-Methode eines Java Programms
- In Programmbeispielen haben wir bereits die `main`-Methode gesehen. Sie ermöglicht das selbständige Ausführen eines Programmes.
- Der Aufruf `java KlassenName` führt die `main`-Methode der Klasse `KlassenName` aus (bzw. gibt eine Fehlermeldung falls, diese Methode nicht existiert).
- Die `main`-Methode hat immer einen Parameter, ein `String`-Array, meist als Eingabe-Variablen `args`. Dies ermöglicht das Verarbeiten von Argumenten, die über die Kommandozeile übergeben werden.

- Der Aufruf

  `java KlassenName <Eingabe1> <Eingabe2> … <Eingabe_n>`

  „füllt“ das `String`-Array (Annahme, der Eingabeparameter heißt `args`) automatisch mit den Eingaben:

```
args[0] = <Eingabe1>
args[1] = <Eingabe2>
…
args[n-1] = <Eingabe_n>
```

- Beispiel für einen Zugriff der `main`-Methode auf das Parameterarray:

```
public class Gruss
{
  public static void gruessen(String gruss)
  {
    System.out.println(gruss);
  }

  public static void main(String[] args)
  {
    gruessen(args[0]);
  }
}
```

- Dadurch ist eine vielfältigere Verwendung möglich:
  - `java Gruss "Hello, World!"`
  - `java Gruss "Hallo, Welt!"`
  - `java Gruss "Servus!"`

Übersicht

- Klassen (so wie wir sie bisher kennen: mit statischen Methoden und Klassenvariablen) sind „ausführbar, wenn sie eine `main`-Methode enthalten
- Klassen können aber auch keine `main`-Methode enthalten, dann stellen sie offenbar verschiedene Algorithmen (in Form anderer statischer Methoden) zur Verfügung.
- Beispiel: die Klasse `Kreis` (siehe Folie 140) stellt die globale Konstante `PI` und die Methoden `kreisUmfang` und `kreisFlaeche` zur Verfügung.
- Klassen stellen ein wichtiges *Strukturierungskonzept* in Java dar:
  - Algorithmen (Methoden) und globale Größen, die konzeptionell zusammen gehören, werden innerhalb einer Klasse definiert.
  - Andere Klassen/Programme können diese Algorithmen dann benutzen
- Das Strukturierungskonzept der Klasse wird oft auch *Modul* genannt und gibt es so (oder so ähnlich) in den meisten Programmiersprachen

- Bei großen Programmen entstehen viele Klassen (und sog. Schnittstellen, die wir später kennenlernen).
- Um einen Überblick über diese Menge zu bewahren, wird ein zusätzliches Strukturierungskonzept benötigt, das von den Details abstrahiert und die übergeordnete Struktur verdeutlicht.
- Ein solches weiteres, übergeordnetes Strukturierungskonzept stellen die *Pakete* (*packages*) dar. Packages erlauben es, Komponenten zu größeren Einheiten zusammenzufassen.
- Die meisten Programmiersprachen bieten dieses Strukturierungskonzept (teilweise unter anderen Namen) an.
- Packages gruppieren Klassen, die einen gemeinsamen Aufgabenbereich haben.

- Packages dienen in Java dazu,
  - große Gruppen von Klassen, die zu einem gemeinsamen Aufgabenbereich gehören, zu bündeln,
  - potentielle Namenskonflikte zu vermeiden,
  - Zugriffe und Sichtbarkeit zu definieren und kontrollieren,
  - eine Hierarchie von verfügbaren Komponenten aufzustellen.
- Jede Klasse in Java ist Bestandteil von genau einem Package.
- Ist eine Klasse nicht explizit einem Package zugeordnet, dann gehört es implizit zu einem *Default*-Package.
- Packages sind hierarchisch gegliedert, können also Unterpackages enthalten, die selbst wieder Unterpackages enthalten, usw.
- Die Package-Hierarchie wird durch Punktnotation ausgedrückt:

```
package.unterpackage1.unterpackage2. ... .Klasse
```

- Der vollständige Name einer Klasse besteht aus dem Klassen-Namen und dem Package-Namen (konventionell aus Kleinbuchstaben): `packagename.KlassenName`
- Um eine Klasse verwenden zu können, muss angegeben werden, in welchem Package sie sich befindet. Mögliche Alternativen:
  - Beim Aufruf einer Methode einer anderen Klasse wird der volle Namen der Klasse angehängt:

    ```
    packagename.KlassenName.methode(…);
    ```
  - Am Anfang des Programms werden die gewünschten Klassen mit Hilfe einer **`import`**-Anweisung eingebunden:

    ```
    import packagename.KlassenName;
    …
    methode(…);
    ```
- Achtung: Werden zwei Klassen gleichen Namens aus unterschiedlichen Packages importiert, müssen die Klassen trotz **`import`**-Anweisung mit vollem Namen aufgerufen werden!

- Klassen des Default-Packages können ohne explizite **`import`**-Anweisung bzw. ohne vollen Namen verwendet werden.
- Wird in der **`import`**-Anweisung eine Klasse angegeben, wird genau diese Klasse importiert. Alle anderen Klassen des entsprechenden Packages bleiben unsichtbar.
- Will man alle Klassen eines Packages auf einmal importieren, kann man dies mit der folgenden **`import`**-Anweisung:

```
import packagename.*;
```

- Achtung: Es werden dabei wirklich nur die Klassen aus dem Package `packagename` eingebunden und nicht etwa auch die Klassen aus Unter-Packages von `packagename`.

- Ein eigenes Package `mypackage` wird angelegt, indem man vor eine Klassendeklaration und vor den **`import`**-Anweisungen die Anweisung **`package`** `mypackage;` platziert.
- Es können beliebig viele Klassen (jeweils aber mit unterschiedlichen Namen) mit der Anweisung **`package`** `mypackage;` im selben Package gruppiert werden.
- Um Namenskollisionen bei der Verwendung von Klassenbibliotheken unterschiedlicher Hersteller zu vermeiden, ist es Konvention, für Packages die URL-Domain der Hersteller in umgekehrter Reihenfolge zu verwenden, z.B.

  `com.sun.` … für die Firma Sun,

  `de.lmu.ifi.dbs.` … für den DBS-Lehrstuhl an der LMU.

- Wie bereits erwähnt, muss die Deklaration einer Klasse `X` in eine Datei `X.java` geschrieben werden.
- Darüberhinaus müssen alle Klassendeklarationen (also die entsprechenden `.java`-Dateien) eines Packages `p` in einem Verzeichnis `p` liegen.
- Beispiel:
  - Die Datei `Klasse1.java` mit der Deklaration der Klasse `package1.Klasse1` liegt im Verzeichnis `package1`.
  - Die Datei `Klasse2.java` mit der Deklaration der Klasse `package1.unterpackage1.Klasse2` liegt im Verzeichnis `package1/unterpackage1`.

- Wir hatten bereits ein Schlüsselwort zur Spezifikation der Sichtbarkeit von Klassen und Klassen-Elementen (globale Variablen / statische Methoden) kennengelernt : **`public`**
  - Klassen und Elemente mit der Sichtbarkeitspezifiktion **`public`** sind von allen anderen Klassen (insbesondere auch Klassen anderer Packages) sichtbar und zugreifbar.
- Darüberhinaus gibt es weitere Möglichkeiten; eine davon ist **`private`**
  - Klassen und Elemente mit der Sichtbarkeitsspezifikation **`private`** sind nur innerhalb der eigenen Klasse (also auch nicht innerhalb möglicher anderer Klassen des selben Packages) sichtbar und zugreifbar.
- Klassen und Elemente, deren Sichtbarkeit *nicht* durch ein entsprechendes Schlüsselwort spezifiziert ist, erhalten per Default die sogenannte *package scoped* (*friendly*) Sichtbarkeit: Diese Elemente sind nur für Klassen innerhalb des selben Packages sichtbar und zugreifbar.
- Bemerkung: es gibt eine weitere Möglichkeit (**`protected`**), die wir mal wieder später kennenlernen …

- Beispiel: Datei `Kreis.java` im Verzeichnis `kreis`

```
package keis;
 public class Kreis {
      public static final double PI = 3.14159;

      public static double kreisUmfang(double radius) {
              return 2 * PI * radius;
      }

      public static double kreisFlaeche(double radius) {
              return PI * radius * radius;
      }
}
```

- Beispiel: Datei `KreisRechnung.java` im Verzeichnis `test`

```
package test;
import kreis.Kreis;

public class KreisRechnung {

      public static void main(String[] args) {
         System.out.println(“Mit pi=“+Kreis.PI+” ergeben sich folgende Werte:);
         System.out.println();

         for(int 1 0; i<=10;i++) {
                double radius = (double) i;
                double flaeche = kreisFlaeche(radius);
                double umfang = kreisUmfang(radius);
                System.out.println(“Radius:”+radius+” ergibt:)
                System.out.println(“   Flaeche: “+flaeche);
                System.out.println(“   Umfang:  “+umfang);
                System.out.println(-------------------------);
          }
      }
}
```

4.6 Algorithmen in Java

4.6.2 Zusammenfassung

- Java implementiert grundsätzlich eine imperative Auffassung von Algorithmen, d.h. Algorithmen als Mengen von Anweisungen, die nacheinander abgearbeitet werden und dabei Zustände verändern
  - Java stellt eine Menge an Grunddatentypen und Operatoren über diese Datentypen als Sorten/Operatoren zur Verfügung
  - Ausdrücke können über diese Operatoren , die Elemente der Grunddatentypen sowie typisierten Variablen gebildet werden
  - Anweisungen können u.a. aus Ausdrücken mit Nebeneffekten, Variablendeklarationen, Verzweigungen, Iterationen, etc. gebildet werden; mehrere Anweisungen können zu Blöcken zusammengefasst werden
  - Algorithmen werden als (statische) Methoden notiert, dadurch wird von den Eingabedaten abstrahiert, die Algorithmen sind beliebig wiederverwendbar; ein Methodenaufruf ist auch wieder eine Anweisung
  - Klassen stellen Funktionalitäten durch Methoden und globale Variablen zur Verfügung; eine Klasse, die eine `main`-Methode enthält ist ausführbar
  - Packages dienen zur Strukturierung meherer Klassen.