# JAVA I

## *Grundlagen der Programmierung*

**Termin 3 von 5:**

## Arrays & Collections

**Letzte Änderung: 17**.03.2017

# Termin 3 – Arrays und Collections

1. **Felder**
2. **Die Klasse „`Arrays`“**
3. **Collections**
4. **Aufzählungen**
5. **DesignPattern: „MVC“ – Model View Control**

# 1. Felder

**Eine Variable**


**Feld:** ***Index*** 0 1 2 3 4

***Wert***

# 1. Felder

## 1.1. Arrays als Sammlung

- Ein Array ist **geordnete Sammlung** von Elementen **desselben Datentyps**
- Neben Datentypen können auch **Objekte derselben Klasse** enthalten sein
- Die **Größe eines Arrays** kann nach der Initialisierung **nicht verändert** werden
- **Arrays sind Objekte** – daher haben sie unter anderem das **Attribut length** ( = Anzahl Elemente)

```
// Ein Array, dass 10 Zahlen aufnehmen kann
int[] numbers;
numbers = new int[10];
int size = numbers.length;     // length (=10) ist ein Attribut für die Arraygroesse

// Ein Array, dass 3 Autos aufnehmen kann
Car[] myCars = new Car[3];
```

# 1. Felder

## 1.2. Arrays initialisieren und auf Elemente zugreifen

```
int[] a1 = new int[3];
a1[0] = 1;  // Die Durchnummerierung beginnt mit 0!
a1[1] = 2;
a1[2] = 3;
a1[3] = 4; // FEHLER das Array hat keinen 4ten Platz

int firstElement = a1[0];  // Beispiel für einen Zugriff

// Beispiele für kompakte Initialisierungen:
int[] a2 = { 4, 5, 6 };
int[] a3 = new int[] { 7, 8, 9 };
```

# 1. Felder

## 1.3. Elemente hinzufügen und entfernen

- Ein Array kann **nicht** vergrößert/verkleinert werden
- Daher muss bei solchen Veränderungen ein neues Array angelegt werden

```
int a1[] = new int[3];       // das Ausgangsarray
[…]
int a2[] = new int[4];       // das „neue" Array, um einen Platz größer

// zunächst muessen alle Elemente kopiert werden
for (int i = 0, i < a1.length; i++) {
          a2[i] = a1[i];
}

a2[3] = 7;          // nun kann ein neues Element hinzugefügt werden
```

# 1. Felder

## 1.4. Die foreach Schleife

- Für Arrays gibt es eine verkürzte Version der for-Schleife
- Dabei werden die Werte aller Elemente **nacheinander als Kopie** bereit gestellt

```
int a1[] = new int[3];

// dieser Ausdruck kann / muss zur Initialisierung verwendet werden
for (int i = 0, i < a1.length; i++) {
        a1[i] = i;
}

// Eine Abfrage kann in dieser Form abgekuerzt werden
for (int element : a1) {
        System.out.println(element);
}
```

# 1. Felder

### 1.5. Mehrdimensionale Arrays

- Mehrdimensionale Arrays werden auch **geschachtelten Arrays** genannt.
- Man könnte geschachtelten Arrays mit einer *Tabelle* vergleichen
- Eine mathematische Betrachtung wäre eine **Matrix**
- Der **Zugriff** geschieht durch Angabe **aller Indizes**
- Beispiele für Arrays mit *2 Zeilen und 3 Spalten* bzw. eine **2x3-Matrix:**

```
int[][] a1 = new int [2][3];
int lastElement = a1[1][2];   // Nicht initialisiert => default = 0!

// Beispiel für eine kompakte Initialisierung
int[][] a2 = {{1, 2, 3 }, {4, 5, 6 }};
```

# 2. Die Klasse „`Arrays`“

## 2.1. Ein Array mit dem selben Wert befüllen

`Arrays.fill(arr, value);`

```
int a1[] = new int[3];

// der manuelle Weg:
for (int i = 0, i < a1.length; i++) {
      a1[i] = 7;
}

// mit der Arrays-Klasse:
Arrays.fill(a1, 7);
```

# 2. Die Klasse „`Arrays`“

## 2.2. Zwei Arrays auf Gleichheit prüfen

`Arrays.equals(arr1, arr2);`

```
// gegeben seien zwei Arrays a1 und a2 - der manuelle Weg:
boolean areEqual = false;
if (a1.length == arr2.length) {
          areEqual = true;
          for (int i = 0, i < a1.length; i++) {
                    if (a1[i] != a2[i]) {
                              areEqual = false;
                              break;
                    }
          }
}
// mit der Arrays-Klasse:
boolean areEqual = Arrays.equals(a1, a2);
```

# 2. Die Klasse „`Arrays`“

**2.3. Die Elemente aufsteigend sortieren** `Arrays.sort(arr);`

- Der manuelle Weg ist etwas komplizierter, doch eine geläufige Übung…

```
int a2[] = new int[4] { 8, 1, 4, 7 };

// mit der Arrays-Klasse:
Arrays.sort(a2);

// Jetzt steht im Array [1, 4, 7, 8]
```

# 3. Collections

**3.1. Was sind Collections?**

- Sammlungen von Objekten, auch Klassen-“Familien“
- vergleichbar mit Arrays (intern werden Arrays verwendet)
- Java hat verschiedene Collection-Arten, unter anderem:
    - **Listen** haben, wie Arrays, eine **Reihenfolge** und einen **int-Index**
    - **Sets** haben ***keine Reihenfolge*** und Elemente sind **einmalig**
    - **Maps** haben einem **frei wählbaren Index**, der zum Beispiel vom Typ String sein kann
- Für jede Collection-Art folgen nun ein Beispiel, mit einem **Auszug** der verwendbaren Methoden

# 3. Collections

### 3.2. Die Listenklasse `ArrayList` (1/2)

```
public boolean isEmpty()                // prueft, ob Liste leer
public int size()                       // gibt die Laenge zurück (wie Array.length)
public void clear()                     // löscht alle Elemente

public boolean add(E e)                 // fuegt ein Element am Ende hinzu
public boolean remove(int index)        // loescht das Element an dieser Stelle
public E get(int index)                 // gibt Element an diesem index zurueck
```

- **`E`** steht für die Klasse des Elements in der Liste.
- Der Rückgabewert von **`getSize()`** kann für **for-Schleife** genutzt werden, um alle Elemente durchzugehen.
- Jedes Element kann auch **`null`** sein!

# 3. Collections

### 3.2. Die Listenklasse `ArrayList` (2/2)

```
ArrayList list = new ArrayList(); // Liste wird initialisiert
list.add(new Car());              // zwei Autos werden erstellt...
list.add(new Car());              //      ...und der Liste hinzugefuegt

int length = list.size();         // gibt die Ganzzahl „2“ zurück
list.remove(0);                   // loescht das ERSTE Autos wieder
Car car = (Car) list.get(0);      // das ursprünglich ZWEITE Auto!
list.clear();                     // loescht alle Autos wieder
boolean empty = list.isEmpty();   // true, da alle Autos weg sind...
```

# 3. Collections

### 3.3. Die Setklasse `HashSet` (1/2)

```
public boolean isEmpty()          // prueft, ob Liste leer
public int size()                 // gibt die Laenge zurück (wie Array.length)
public void clear()               // löscht alle Elemente

public boolean add(E e)           // fuegt ein Element am Ende hinzu
public boolean remove(E e)        // loescht das Element, wenn vorhanden
public Iterator E iterator()      // alle Elemente durchlaufen
```

- **Ein Iterator hat folgende Methoden:**

```
public boolean hasNext()          // prueft ob es noch mind. Ein weiteres Element gibt
public E next()                   // gibt das naechste Element zurueck
public void remove()              // loescht das Element, das mit "next()" aufgerufen wurde
```

# 3. Collections

### 3.3. Die Setklasse `HashSet` (2/2)

```
HashSet set = new HashSet();          // Set wird initialisiert
set.add(new Car());                   // Auto wird hinzugefuegt

Car myCar = new Car();
set.add(myCar);                       // mein Auto wird hinzugefuegt
set.add(myCar);                       // NIX passiert - Set kennt mein Auto schon!

Iterator it = set.iterator();                 // Iterator wird angefordert
while (it.hasNext()) {                        // typische Iterator-schleife
       Car currentCar = (Car) it.next();      // nacheinander alle Autos
}
```

# 3. Collections

### 3.4. Die Mapklasse `HashMap` (1/2)

```
public boolean isEmpty()                    // prueft, ob Liste leer
public int size()                           // gibt die Laenge zurück (wie Array.length)
public void clear()                                 // löscht alle Elemente

public boolean put(K key, V value e)                // fuegt ein Element mit neuem Schluessel hinzu
public boolean remove(K key)                        // loescht das Element mit diesem Schluessel
public V get(K key)                                 // Zugriff auf das Element mit diesem Schluessel

public Collection values()                  // alle Elemente als Collection (Set) zurückgeben
public Map.Entry entrySet()                 // gibt Schluessel und Wert zurück (naechste Folie!)
```

- Auf die „`vales()`“ kann wiederum ein Iterator wie bei einem Set angewendet werden

# 3. Collections

### 3.4. Die Mapklasse HashMap (2/2)

```
HashMap<String, Car> map;            // Map wird definiert
map = new HashMap <String, Car>();   // Map wird initialisiert

map.put(„Steffen", new Car());       // neues Auto mit Schluessel „Steffen"
map.put(„Lykka", new Car());         // neues Auto mit Schluessel „Lykka"
map.put(„Lykka", new Car());         // Lykka hat ein neues Auto...
                                     //      ...das alte wurde zerstört

for (Map.Entry e : map.entrySet()) {          // typische Iterator-schleife
      String key = (String) e.getKey();       // Zugriff auf aktuellen Schluessel
      Car currentCar = (Car) e.getValue(); // nacheinander alle Autos...
}
```

# 3. Collections

**3.5. Umwandlung Array <=> Collection**

- Es gibt die Möglichkeit, aus jeder Collection ein primitives Array zu machen:

```
Car[] cars = listOfCars.toArray();
```

- Die einzelnen Collections können ebenfalls Arrays in ein Objekt ihrer Art umwandeln:

```
Collections.addAll(myCollection, myArray);
```

- **`myCollection`** kann z.B. ein **`Set`**, eine **`List`** oder eine **`Map`** sein

- Meist macht es aber schon bei der **Planung** Sinn zu überlegen: **Welche Collection? Oder doch ein Array?**

# 4. Aufzählungen

Der Aufzählungstyp „enum“ ist ein spezieller Datentyp, mit der eine Variable auf einen vordefinierten konstanten Wert gesetzt werden kann.

Typische Beispiele sind die Himmelsrichtungen (NORTH, SOUTH, EAST, WEST) oder die Tage der Woche:

```
public enum Day {
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
        THURSDAY, FRIDAY, SATURDAY
}
```

# 4. Aufzählungen

Durch ein **enum** wird Code eindeutiger und einfacher lesbar:

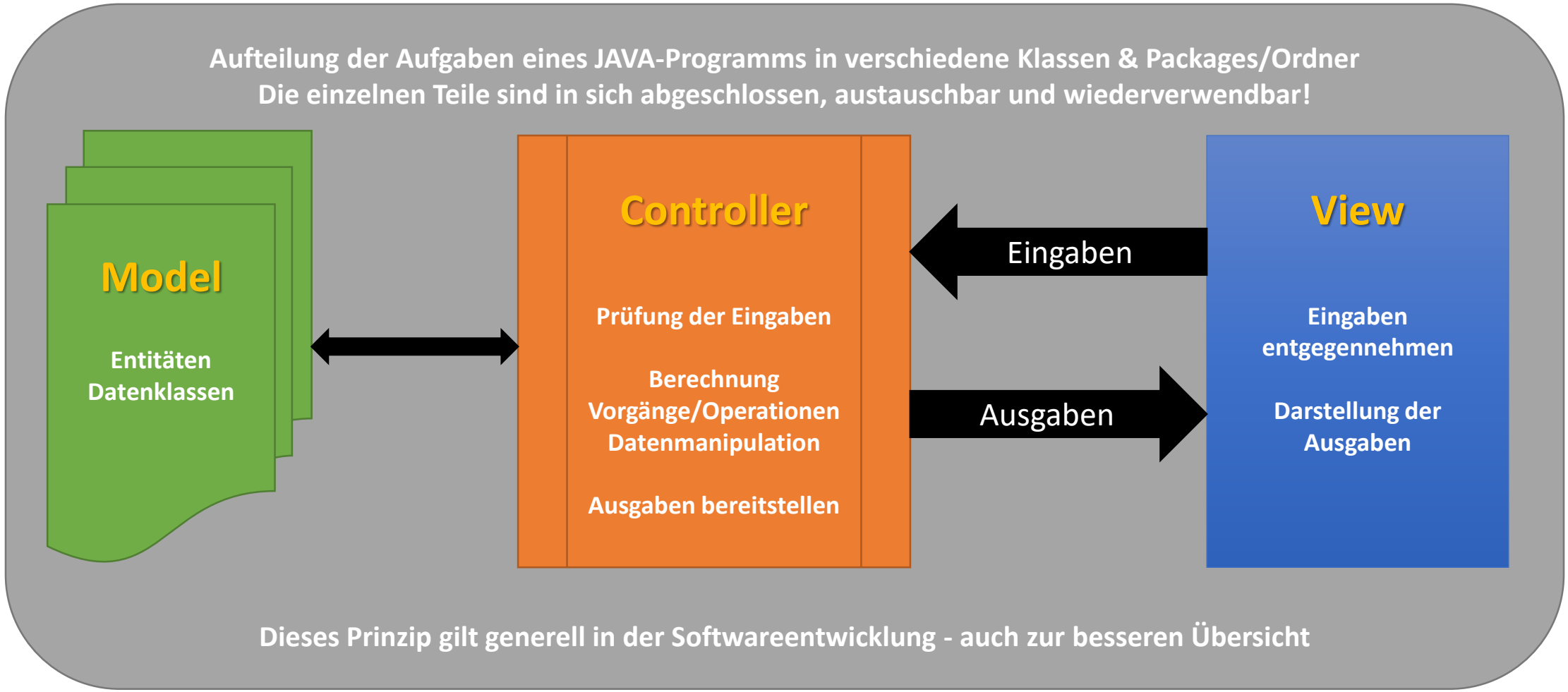
```
Day day = Day.MONDAY;          // Zuweisungsbeispiel

switch(day) {                  // Anwendungsbeispiel
      case MONDAY:
            System.out.println(„Los geht’s!“);
            break;
      case SATURDAY: case SUNDAY:
            System.out.println(„leg dich wieder hin…“);
            break;
}
```

# 4. Aufzählungen

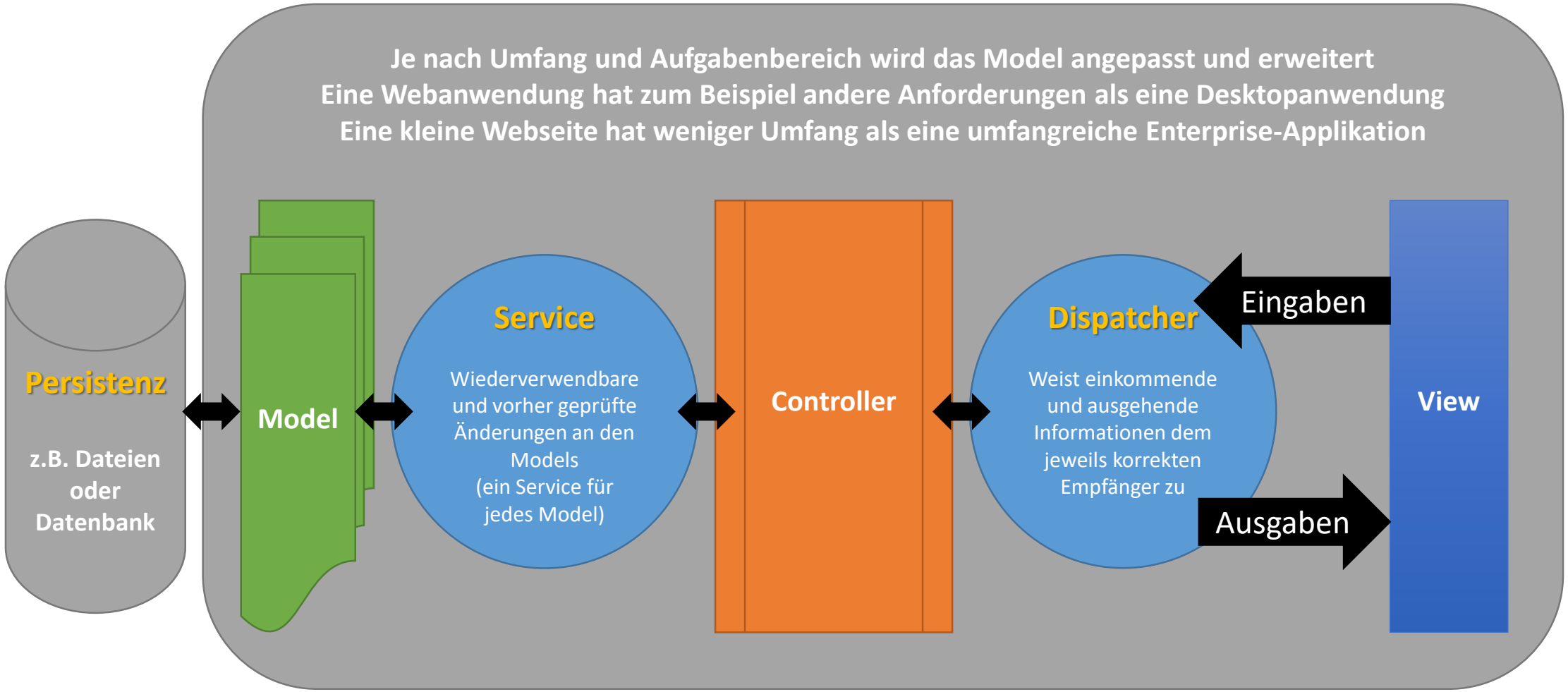
**BONUS:** Wie alles sind auch enums in JAVA Klassen, daher ergeben sich weitere Möglichkeiten es um Informationen zu erweitern:

```
public enum Day {
        SUNDAY („Sonntag"),
        ...       // der Rest der Woche
        SATURDAY („Samstag");

        // die zugehörigen Konstanten sind konstante Member
        private final String de;
        // Konstruktor muss folgende Form haben, wird automatisch verarbeitet
        Day(String de) { this.de = de; }
        // Zugriff auf den jeweils konstanten Wert
        public String de() { return de; }
}

System.out.println(„Wir sehen uns " + Day.TUESDAY.getDe());
```

# 5. DesignPattern: „MVC“ – Model View Control



# 5. DesignPattern: „MVC“ – Model View Control



# 5. DesignPattern: „MVC“ – Model View Control