Universität
Osnabrück

Informatik B – Objektorientierte Programmierung in Java

Vorlesung 02:
Objektorientierte Programmierung (Teil 2)

Inhalt (1)

- Konstruktoren
- Destruktoren
- Beispiel
  - Konto
- Ausgewählte Klassen
  - String

Inhalt (2)

- Klassen mit static-Elementen
  - Klassenattribute
  - Konstanten
  - Klassenoperationen
  - Statische Konstruktoren
- Beispiel
  - ZählerTest
  - MaxTest
- Ausgewählte Klassen
  - System

Konstruktoren

- In jeder objektorientierten Programmiersprache lassen sich spezielle Operationen definieren, die bei der Initialisierung eines Objekts aufgerufen werden: die *Konstruktoren*.
- In Java werden Konstruktoren als Operationen ohne Rückgabewert definiert, die den Namen der Klasse erhalten, zu der sie gehören.
- Konstruktoren dürfen eine beliebige Anzahl an Parametern haben und können überladen werden.

Konstruktoren

- Die Erweiterung der Klasse **Auto** um einen Konstruktor, der den Namen des **Auto**-Objekts vorgibt, sieht beispielsweise so aus:

```
class Auto
{
   String name;
    int erstzulassung;
    int leistung;

    Auto(String name)
    {
       this.name = name;
    }
}
```


Konstruktoren

- Soll ein Objekt unter Verwendung eines parametrisierten Konstruktors instantiiert werden, so sind die Argumente wie bei einem Operationsaufruf in Klammern nach dem Namen des Konstruktors anzugeben:

```
Auto dasAuto = new Auto("Porsche 911");
System.out.println(dasAuto.name);
```


Konstruktoren

- In diesem Fall wird zunächst Speicher für das Auto-Objekt beschafft und dann der Konstruktor aufgerufen.
- Dieser initialisiert seinerseits das Attribut **name** mit dem übergebenen Argument **"Porsche 911"**.
- Der nachfolgende Aufruf schreibt dann diesen Text auf den Bildschirm.

Konstruktoren

- Explizite Konstruktoren werden immer dann eingesetzt, wenn zur Initialisierung eines Objektes besondere Aufgaben zu erledigen sind.
- Es ist dabei durchaus gebräuchlich, Konstruktoren zu überladen und mit unterschiedlichen Parameterlisten auszustatten.
- Beim Ausführen der **new**-Anweisung wählt der Compiler anhand der aktuellen Parameterliste den passenden Konstruktor aus und ruft ihn mit den angegebenen Argumenten auf.

Konstruktoren

- Das vorige Beispiel wird nun um einen Konstruktor erweitert, der alle Attribute initialisiert:

Konstruktoren

```
class Auto
{
    String name;
   int erstzulassung;
   int leistung;

   Auto(String name)
   { this.name = name;
   }

   Auto(String name, int erstzulassung, int leistung)
   { this.name = name;
     this.erstzulassung = erstzulassung;
     this.leistung = leistung;
   }
}
```


Konstruktoren

- Default-Konstruktor
  - Falls eine Klasse überhaupt keinen expliziten Konstruktor besitzt, so wird beim Anlegen eines Objektes ein parameterloser default-Konstruktor aufgerufen.
  - Definiert die Klasse dagegen einen eigenen parameterlosen Konstruktor, überlagert dieser den default-Konstruktor und wird bei allen parameterlosen Instantiierungen dieses Objekts verwendet.

Konstruktoren

- Verkettung von Konstruktoren
  - Konstruktoren können in Java verkettet werden, d.h. sie können sich gegenseitig aufrufen. Der aufzurufende Konstruktor wird dabei als eine normale Operation angesehen, die über den Namen **`this`** aufgerufen werden kann.
  - Die Unterscheidung zum bereits vorgestellten **`this`**-Zeiger nimmt der Compiler anhand der runden Klammern vor, die dem Aufruf folgen.
  - Der im vorigen Beispiel vorgestellte Konstruktor hätte damit auch so geschrieben werden können:

Konstruktoren

```
class Auto
{
   String name;
    int erstzulassung;
    int leistung;

    Auto(String name)
    { this.name = name;
    }

    Auto(String name, int erstzulassung, int leistung)
    { this(name);
      this.erstzulassung = erstzulassung;
      this.leistung = leistung;
    }
}
```


Konstruktoren

- Der Vorteil der Konstruktorenverkettung besteht darin, dass vorhandener Code wiederverwendet wird.
- Führt ein parameterloser Konstruktor eine Reihe von nichttrivialen Anweisungen durch, so ist es natürlich sinnvoller, diesen in einem spezialisierteren Konstruktor durch Aufruf wiederzuverwenden, als den Code zu duplizieren.
- Wird ein Konstruktor in einem anderen Konstruktor derselben Klasse explizit aufgerufen, muss dies als erste Anweisung innerhalb des Konstruktors geschehen.
- Steht der Aufruf eines anderen Konstruktors nicht an erster Stelle in einem Konstruktor, gibt es einen Compiler-Fehler.

Beispiel

- Kontoverwaltung

Destruktoren

- Neben Konstruktoren, die während der Initialisierung eines Objekts aufgerufen werden, gibt es in Java auch *Destruktoren*.
- Sie werden unmittelbar vor dem Zerstören eines Objekts aufgerufen.

Destruktoren

- Definition eines Destruktors
  - Ein Destruktor wird als parameterlose Operation mit dem Namen **`finalize`** definiert:

```
protected void finalize()
{
   ...
}
```


Destruktoren

- Da Java über eine automatische Speicher-verwaltung verfügt, kommt den Destruktoren eine viel geringere Bedeutung zu als in anderen objektorientierten Sprachen.
- Anders als etwa in C++ muss sich der Entwickler ja nicht um die Rückgabe von belegtem Speicher kümmern;
  und das ist sicher eine der Hauptaufgaben von Destruktoren in C++.

Destruktoren

- Tatsächlich garantiert die Sprachspezifikation von Java nicht, dass ein Destruktor überhaupt aufgerufen wird.
- Wenn er aber aufgerufen wird, so erfolgt dies nicht, wenn die Lebensdauer des Objektes endet, sondern dann, wenn der Garbage Collector den für das Objekt reservierten Speicherplatz zurückgibt.
- Dies kann unter Umständen nicht nur viel später der Fall sein (der Garbage Collector läuft ja als asynchroner Hintergrundprozess), sondern auch gar nicht.

Destruktoren

- Wird nämlich das Programm beendet, bevor der Garbage Collector das nächste Mal aufgerufen wird, werden auch keine Destruktoren aufgerufen.
- Selbst wenn Destruktoren aufgerufen werden, ist die Reihenfolge oder der Zeitpunkt ihres Aufrufs undefiniert.
- Der Einsatz von Destruktoren in Java sollte also mit der nötigen Vorsicht erfolgen.

Ausgewählte Klassen

- String
  - Die Klasse **`String`** repräsentiert Zeichenketten, bestehend aus Unicode-Zeichen.
  - Objekte vom Typ **`String`** (im Folgenden kurz "String" genannt) sind nach der Initialisierung nicht mehr veränderbar.
  - Alle Stringliterale (konstante Zeichenketten) werden als Objekte dieser Klasse implementiert.
  - So zeigt z.B. die Referenzvariable **`name`** nach der Initialisierung durch **`String name = "Schmitz"`** auf das Objekt, das die konstante Zeichenkette "Schmitz" repräsentiert.

Ausgewählte Klassen

- String
  - Konstruktoren:
    - **`String()`**
      erzeugt ein String-Objekt, das keine Zeichen enthält.
    - **`String(String s)`**
      erzeugt ein String-Objekt mit dem Inhalt von s.
    - **`String(char [] c)`**
      erzeugt ein String-Objekt, in das alle **`char`**-Zeichen des Arrays **`c`** übernommen werden.
  - Wichtige Operationen der Klasse String
    - **`int length()`**
      liefert die Länge der Zeichenkette.

Ausgewählte Klassen

- String
  - Verketten
    - **`String concat(String s)`**
      erzeugt einen neuen String, der den Inhalt von **`s`** an den des Strings, für den die Operation aufgerufen wurde, anhängt.
    - Strings können auch mit dem Operator **`+`** verkettet werden. Ein String-Objekt entsteht auch dann, wenn dieser Operator auf einen String und eine Variable oder ein Literal vom primitiven Datentyp angewandt wird. Beispiel:

```
System.out.println("Name.....: "+meinKombi.name);
System.out.println("Zugelassen: "+meinKombi.erstzulassung);
```


Ausgewählte Klassen

- String
  - Extrahieren
    - **`char charAt(int i)`**
      liefert das Zeichen an der Position **`i`** der Zeichenkette. Das erste Zeichen hat die Position **`0`**.
    - **`String substring(int begin)`**
      erzeugt einen String, der alle Zeichen des Strings, für den die Operation aufgerufen wurde, ab der Position **`begin`** enthält.
    - **`String substring(int begin, int end)`**
      erzeugt einen String, der alle Zeichen ab der Position **`begin`** bis **`end - 1`** enthält.
    - **`String trim()`**
      schneidet alle zusammenhängenden Leer- und Steuerzeichen (das sind die Zeichen kleiner oder gleich **`'\u0020'`**) am Anfang und Ende der Zeichenkette ab.

Ausgewählte Klassen

- String
  - Vergleichen
    - **`boolean equals(Object obj)`**
      liefert **`true`**, wenn **`obj`** ein String ist und das aktuelle **`String`**-Objekt und **`obj`** die gleiche Zeichenkette repräsentieren.
    - **`boolean equalsIgnoreCase(String s)`**
      wirkt wie **`equals`**, ignoriert aber eventuell vorhandene Unterschiede in der Groß- und Kleinschreibung.
    - **`boolean startsWith(String s)`**
      liefert **`true`**, wenn die Zeichenkette im aktuellen String mit der Zeichenkette in **`s`** beginnt.
    - **`boolean endsWith(String s)`**
      liefert **`true`**, wenn die Zeichenkette im aktuellen String mit der Zeichenkette in **`s`** endet.

Ausgewählte Klassen

- String
  - Vergleichen
    - **`boolean regionMatches(int i, String s, int j, int l)`**
      liefert **`true`**, wenn der Teilstring des Strings, der ab der Position **`i`** beginnt und die Länge **`l`** hat, mit dem Teilstring von **`s`** übereinstimmt, der an der Position **`j`** beginnt und **`l`** Zeichen lang ist.
    - **`int compareTo (String s)`**
      vergleicht diese Zeichenkette mit der Zeichenkette in **`s`** lexikographisch und liefert **`0`**, wenn beide Zeichenketten gleich sind, einen negativen Wert, wenn diese Zeichenkette kleiner ist als die in **`s`**, und einen positiven Wert, wenn diese Zeichenkette größer ist als die in **`s`**.

Ausgewählte Klassen

- String
  - Suchen
    - **`int index0f(int c)`**
      liefert die Position des ersten Auftretens des Zeichens **`c`** in dieser Zeichenkette, andernfalls wird **`-1`** zurückgegeben.
    - **`int index0f(int c, int from)`**
      wirkt wie die vorige Operation mit dem Unterschied, dass die Suche ab der Position **`from`** in dieser Zeichenkette beginnt.
    - **`int index0f(String s)`**
      liefert die Position des ersten Zeichens des ersten Auftretens der Zeichenkette **`s`** in dieser Zeichenkette, andernfalls wird **`-1`** zurückgegeben.
    - **`int index0f(String s, int from)`**
      wirkt wie die vorige Operation mit dem Unterschied, dass die Suche ab der Position **`from`** in dieser Zeichenkette beginnt.

Ausgewählte Klassen

- String
  - Suchen
    - Die folgenden vier Operationen suchen analog nach dem letzten Auftreten des Zeichens **`c`** bzw. der Zeichenkette **`s`**:

      - **`int lastIndex0f(int c)`**
      - **`int lastIndex0f(int c, int from)`**

      - **`int lastIndex0f(String s)`**
      - **`int lastIndex0f(String s, int from)`**

Ausgewählte Klassen

- String
  - Ersetzen
    - **`String replace(char old, char new)`**
      erzeugt einen String, in dem jedes Auftreten des Zeichens **`old`** in der aktuellen Zeichenkette durch das Zeichen **`new`** ersetzt ist.
    - **`String toLowerCase()`**
      wandelt die Großbuchstaben der Zeichenkette in Kleinbuchstaben um und liefert diesen String dann zurück.
    - **`String toUpperCase()`**
      wandelt die Kleinbuchstaben der Zeichenkette in Großbuchstaben um und liefert diesen String dann zurück.

Ausgewählte Klassen

- String
  - Konvertieren
    - **`static String value0f(Typ x)`**
      wandelt **`x`** in einen String um. **`Typ`** steht für **`boolean`**, **`char`**, **`int`**, **`long`**, **`float`**, **`double`**, **`char[]`** und **`Object`**.
      Im letzten Fall wird "**`null`**" geliefert, falls das Objekt **`null`** ist, sonst wird **`x.toString()`** geliefert.

Ausgewählte Klassen

- String
  - Beispielprojekt StringTest

Klassen mit static-Elementen

- Java ist eine konsequent objektorientierte Sprache, in der es weder globale Funktionen noch globale Variablen gibt.
- Da es aber mitunter sinnvoll ist, Operationen aufzurufen, die nicht an Objekte einer Klasse gebunden sind, haben die Sprachdesigner den Modifikator **`static`** für Operationen und Variablen eingeführt.

Klassenattribute

- Attribute, die mit dem Modifikator **`static`** versehen werden, nennt man Klassenvariablen oder *Klassenattribute*.
- Im Gegensatz zu den (normalen) Attributen, die an ein Objekt gebunden sind, existieren Klassenattribute unabhängig von einem Objekt.
- Jedes Klassenattribut wird nur einmal angelegt und kann von allen Operationen der Klasse benutzt werden.
- Da sich alle Objekte die Variable "teilen", sind Veränderungen, die ein Objekt vornimmt, auch in allen anderen Objekten der Klasse sichtbar.

Klassenattribute

- Klassenattribute sind daher vergleichbar mit globalen Variablen, denn ihre Lebensdauer erstreckt sich auf das gesamte Programm.
- Namenskollisionen können allerdings nicht auftreten, denn der Zugriff von außen erfolgt durch Qualifizierung mit dem Klassennamen in der Form ***`Klassenname.Variablenname`***.

Klassenattribute

- Ein Beispiel für die Verwendung eines Klassenattributes besteht darin, einen Objektzähler in eine Klasse einzubauen.
- Hierzu wird ein beliebiges Klassenattribut eingeführt, das beim Erzeugen eines Objekts hoch und beim Zerstören heruntergezählt wird.
- Das folgende Beispiel demonstriert das für die Klasse **`Auto`**:

Klassenattribute

Beispiel

```
class Auto
{
  static int objcnt = 0;
  Auto()
  { ++objcnt;
  }
  protected void finalize()
  { --objcnt;
  }
  public static void main(String[] args)
  { Auto auto1;
    Auto auto2 = new Auto();
    System.out.println("Anzahl Auto-Objekte: "+Auto.objcnt);
    auto2.finalize();
    System.out.println("Anzahl Auto-Objekte: "+Auto.objcnt);
  }
}
```

Klassenattribute

- Wie lautet die Ausgabe des Programms?

Konstanten

- Eine andere Anwendung von Klassenattributen besteht in der Deklaration von Konstanten.
- Dazu wird der Modifikator **`static`** mit dem Modifikator **`final`** kombiniert, um eine unveränderliche Variable mit unbegrenzter Lebensdauer zu erzeugen:

```
class Auto
{
  static final double STEUERSATZ = 18.9;
}
```


Konstanten

- Durch die Anwendung von **`final`** wird verhindert, dass der Konstanten **`STEUERSATZ`** während der Ausführung des Programms ein anderer Wert zugewiesen wird.
- Die Konvention, Konstantennamen groß zu schreiben, wurde von der Programmiersprache C übernommen.

Klassenoperationen

- Neben Klassenattributen gibt es in Java auch *Klassenoperationen*, d.h. Operationen, die unabhängig von einem bestimmten Objekt existieren.
- Klassenoperationen werden ebenfalls mit Hilfe des Modifikators **`static`** deklariert und - analog zu Klassenattributen - durch Voranstellen des Klassennamens aufgerufen.

Klassenoperationen

- Klassenoperationen können auf Klassenattribute zugreifen.
- Da Klassenoperationen unabhängig von konkreten Objekten ihrer Klasse existieren, ist ein Zugriff auf (normale) Attribute nicht möglich.
- Diese Trennung äußert sich darin, dass Klassenoperationen keine **`this`**-Referenz besitzen.
- Der Zugriff auf Attribute und der Aufruf von (normalen) Operationen in einer Klassen-operation wird daher schon bei der Übersetzung durch den Compiler als Fehler erkannt.

Klassenoperationen

- Klassenoperationen werden häufig da eingesetzt, wo Funktionalitäten zur Verfügung gestellt werden, die nicht datenzentriert arbeiten oder auf primitiven Datentypen operieren.
- Beispiele für beide Arten sind in der Klassenbibliothek zu finden.
- Zur ersten Gruppe gehören beispielsweise die Operationen der Klasse **`System`**.

Klassenoperationen

- Die Klasse **`System`** ist eine Art Toolbox, die Funktionen für den Aufruf des Garbage Collectors oder das Beenden des Programms zur Verfügung stellt.
- Zur zweiten Gruppe gehören beispielsweise die Operationen der Klasse **`Math`**, die eine große Anzahl an Funktionen zur Fließkommaarithmetik zur Verfügung stellt.
- Da die Fließkommatypen primitiv sind, hätte ein objektbasiertes Design der Operationen an dieser Stelle wenig Sinn gemacht.

Klassenoperationen

- Das folgende Programmbeispiel zeigt die Verwendung der Klassenoperation **`sqrt`** der Klasse **`Math`** zur Ausgabe einer Tabelle von Quadratwurzeln:

```
public static void main(String[] args)
{
  double x, y;
  for (x = 0.0d; x <= 10.0d; x = x + 1.0d)
  {
       y = Math.sqrt(x);
       System.out.println("sqrt("+x+") = "+y);
  }
}
```

Statische Konstruktoren

- Bisher haben Sie nur die Möglichkeit kennen gelernt, Klassenattributen während der Deklaration einen Wert zuzuweisen.
- Falls komplexere Initialisierungen benötigt werden, können zur Initialisierung von Klassenattributen statische Konstruktoren definiert werden.
- Sie ähneln gewöhnlichen Konstruktoren und dienen wie diese dazu, Variablen mit einem definierten Startwert zu belegen.

Statische Konstruktoren

- Im Gegensatz zu einem gewöhnlichen Konstruktor erfolgt der Aufruf eines statischen Konstruktors aber nicht mit jedem neu angelegten Objekt, sondern nur einmal zu Beginn des Programms.
- Ein statischer Konstruktor wird als parameterlose Operation mit dem Namen **`static`** definiert:

Statische Konstruktoren

- Beispiel:

```
class Test
{
   static int i;
   static int j;

   static
   {
      i = 5;
      j = 3 + i;
   }
}
```

  - Da er vom Java-Interpreter aufgerufen wird, wenn die Klasse geladen wird, ist eine benutzerdefinierte Parametrisierung nicht möglich.

Statische Konstruktoren

  - Eine Klasse darf mehr als einen statischen Konstruktor definieren.
  - Die Konstruktoren werden dann in der Reihenfolge ihrer Deklaration aufgerufen.

Beispiel

- ZaehlerTest
- MaxTest

Ausgewählte Klassen

- System
  - Die Klasse **`System`** enthält wichtige Operationen zur Kommunikation mit dem zu Grunde liegenden Betriebssystem.
  - Es können keine Objekte vom Typ **`System`** erzeugt werden.
  - Alle Operationen von **`System`** sind Klassenoperationen.
  - **`System`** enthält folgende Klassenattribute:

```
final static InputStream in
final static PrintStream out
final static PrintStream err
```


Ausgewählte Klassen

- System
  - Sie sind die Standarddatenströme zur Eingabe, Ausgabe und Fehlerausgabe.
  - Die Klassen **`InputStream`** und **`PrintStream`** werden später im Rahmen der Dateiverarbeitung erläutert.
  - Beispiel: **`System.out.println(„Hallo“);`**
  - Operationen:
    - **`static void gC`**
      ruft den Garbage Collector auf
    - **`static void exit(int status)`**
      beendet das laufende Programm. **`status`** wird an den Aufrufer des Programms übergeben. Üblicherweise signalisiert 0 ein fehlerfreies Programmende.

Ausgewählte Klassen

- System
  - Operationen
    - **`static long currentTimeMillis()`**
      liefert die Anzahl Millisekunden, die seit dem 1.1.1970 um 00:00:00 Uhr GMT bis zum Aufruf der Operation vergangen sind.
    - **`static Properties getProperties()`**
      liefert die definierten Systemeigenschaften (System-Properties) der Plattform als Objekt der Klasse **`Properties`**.
    - **`static String getProperty(String key)`**
      liefert den Wert der Systemeigenschaft mit dem Namen **`key`** oder **`null`**, wenn keine Eigenschaft mit diesem Namen gefunden wurde.

Ausgewählte Klassen

- System
  - Operationen:
    - **`static String getProperty(String key, String defaultValue)`**
      liefert den Wert der Systemeigenschaft mit dem Namen **`key`** oder den Wert **`defaultValue`**, wenn keine Eigenschaft mit diesem Namen gefunden wurde.

Ausgewählte Klassen

- System
  - Property — Bedeutung

Property	Bedeutung
`file.separator`	`Dateipfadtrennzeichen`
`java.class.path`	`aktueller Klassenpfad`
`java.class.version`	`Version der Klassenbibliothek`
`java.home`	`Installationsverzeichnis`
`java.vendor`	`Herstellername`
`java.vendor.url`	`URL des Herstellers`
`java.version`	`Java-Versionsnummer`
`line.separator`	`Zeilentrennzeichen`
`os.arch`	`Betriebssystemarchitektur`
`os.name`	`Betriebssystemname`
`os.version`	`Betriebssystemversion`
`path.separator`	`Trennzeichen in PATH-Angaben`
`user.dir`	`aktuelles Arbeitsverzeichnis`
`user.home`	`Home-Verzeichnis`
`user.name`	`Anmeldename`

Beispiel

- SystemTest