6 Das Klassenkonzept in Java

Die Sprache Java ist eine vollständig objektorientierte Sprache. In den bisherigen Kapiteln wurden allerdings keine objektorientierten Themen behandelt, sondern die Grundlagen der strukturierten Programmierung besprochen. Das war notwendig, um eine Basis für die weiteren Themen zu schaffen. Trotzdem waren einige Aspekte bereits objektorientiert, wurden aber nur so weit beschrieben, dass es für die Ausführung eines Programms ausreichte. Mit diesem Kapitel beginnt nun die objektorientierte Programmierung in Java. Darunter kann eine spezielle Art der Programmierung verstanden werden, die versucht, gewisse Gegebenheiten möglichst realitätsnah umzusetzen. Im Mittelpunkt der objektorientierten Programmierung steht das **Objekt** bzw. die **Klasse**. Eine Klasse kann als eine Art Bauplan betrachtet werden, mit dem Objekte gebildet werden können. Die Begriffe Objekt und Klasse werden nun näher betrachtet.

Was ist ein Objekt?
Ein Objekt ist eine softwaretechnische Repräsentation eines realen oder gedachten, klar abgegrenzten Gegenstandes oder Begriffs. Das Objekt erfasst alle Aspekte des Gegenstandes durch Attribute (Eigenschaften) und Methoden.

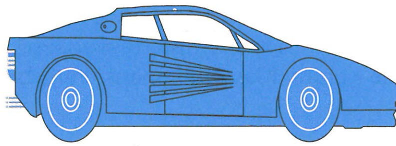
Was sind Attribute und Methoden?
Attribute sind die Eigenschaften des Objektes. Sie beschreiben den Gegenstand vollständig. Attribute sind geschützt gegen Manipulation von außen (das nennt man Kapselung). Methoden beschreiben die Operationen, die mit dem Objekt (bzw. seinen Attributen) durchgeführt werden können. Von außen erfolgt der Zugriff auf Attribute durch die Methoden.

Was ist eine Klasse?
Unter einer Klasse versteht man die softwaretechnische Beschreibung eines Bauplanes für ein Objekt. Aus einer Klasse können dann Objekte abgeleitet (gebildet, instanziiert) werden.

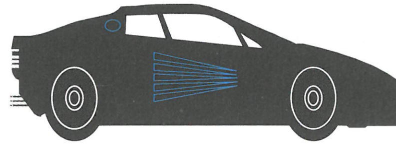
Diese etwas abstrakten, aber wichtigen Begriffsdefinitionen sollen nun anhand von Beispielen veranschaulicht werden.

Beispiel:

Diese Rennwagen sind konkrete Objekte. Sie haben Attribute wie Farbe, Leistung in kW und den Hubraum.

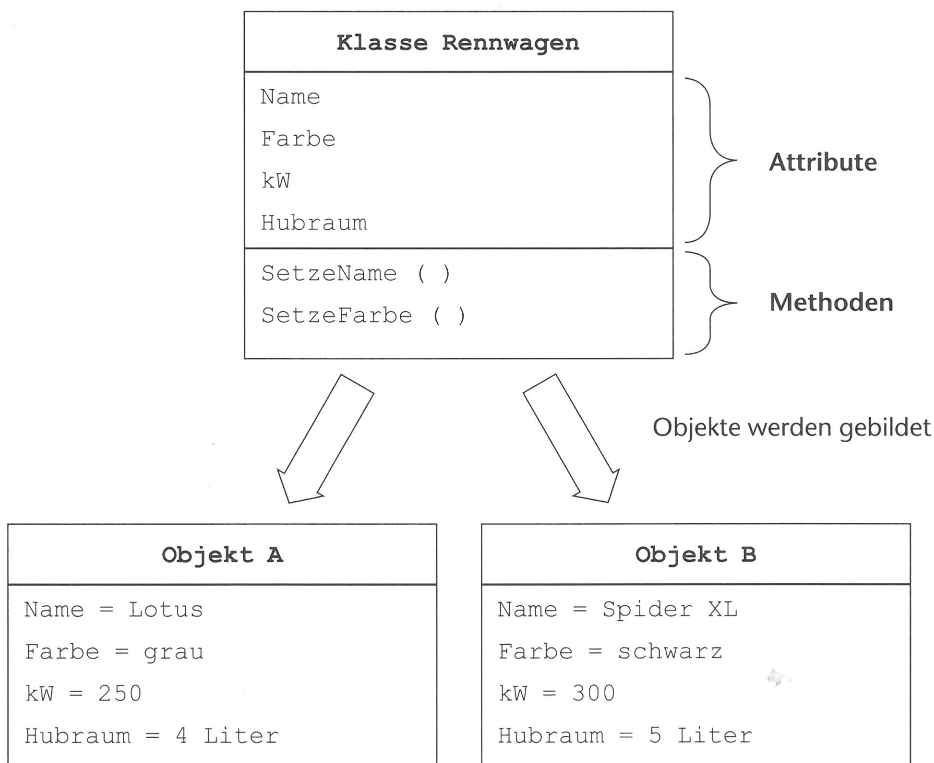


Name:	Lotus
Farbe:	blau
kW:	250
Hubraum:	4 Liter



Name:	Spider XL
Farbe:	schwarz
kW:	300
Hubraum:	5 Liter

Beide Rennwagen haben dieselben Attribute. Sie unterscheiden sich nur in den Attributwerten. Der Spider XL hat beispielsweise eine höhere Leistung als der Lotus. Man könnte sagen, dass beide Rennwagen mithilfe desselben Bauplanes hergestellt worden sind. Der zugrunde liegende Bauplan könnte als **Klasse** `Rennwagen` bezeichnet werden. Die folgende Darstellung der Klassen und Objekte entspricht der allgemeinen Form, um Klassen und Objekte darzustellen.



Hinweise

Die Objektorientierung und die neuen Begriffe erscheinen gerade am Anfang recht abstrakt und es scheint kaum vorstellbar, wie eine neue Software objektorientiert programmiert werden soll. Dagegen hilft nur eins: Schritt für Schritt die Aspekte der objektorientierten Programmierung (**OOP**) kennenlernen und an konkreten Beispielen umsetzen. Gute objektorientierte Programmentwicklung hat auch viel mit Erfahrung zu tun.

Neben der veränderten Sichtweise der Programmierung hat die OOP auch ganz praktische Vorteile gegenüber der strukturierten oder prozeduralen Programmierung. Diese Vorteile sind beispielsweise die Kapselung von Daten in den Objekten oder die Vererbung. Kapselung von Daten bedeutet, dass der Zugriff auf die Attribute eines Objektes kontrolliert abläuft. Dieser kontrollierte Zugriff geschieht über die Methoden eines Objektes. Dadurch wird beispielsweise verhindert, dass ein wichtiges Attribut eines Objektes aus Versehen mit einem falschen Wert

beschrieben wird. Die Vererbung erspart dem Programmierer ungemein viel Arbeit, weil er einmal geschriebene Klassen an andere Klassen vererben kann. Das komplette Konzept der OOP wird allerdings dann erst richtig deutlich, wenn die Kapitel Klassenkonzept, Vererbung und Polymorphismus bearbeitet wurden.

6.1 Die erste Klasse in Java

In diesem Kapitel geht es hauptsächlich um die konkrete Umsetzung einer Klasse in Java. Zuerst wird der allgemeine Aufbau einer Klasse beschrieben. Dabei stehen vor allem die Attribute und deren Sichtbarkeit im Vordergrund. Das steht im unmittelbaren Zusammenhang mit einem wichtigen Aspekt der OOP, der Kapselung. Anschließend werden Funktionsweise und Aufbau von Methoden beleuchtet.

6.1.1 Aufbau einer Klasse in Java

Eine Klasse in Java wird mit dem Schlüsselwort **`class`** eingeleitet. Innerhalb einer Klasse (eingerahmt durch geschweifte Klammern) gibt es Attribute und Methoden, die mit einem Sichtbarkeitsmodifizierer (`private`, `public` oder `protected`) versehen werden. Diese einzelnen Modifizierer haben unterschiedliche Auswirkungen.

Der private-Modifizierer:
Alle Attribute (und auch Methoden), die damit gekennzeichnet werden, sind von außen nicht zugreifbar. Der Zugriff kann nur über geeignete (öffentliche) Methoden erfolgen.

Der public-Modifizierer:
Alle Methoden (und auch Attribute), die damit gekennzeichnet werden, sind von außen zugreifbar. Diese Elemente bezeichnet man auch als Schnittstelle der Klasse nach außen. Die Kommunikation mit der Klasse (bzw. mit einem Objekt dieser Klasse) findet über diese Schnittstelle (`public`-Elemente) statt.

Der protected-Modifizierer:
Dieser Modifizierer verhält sich nach außen wie der `private`-Modifizierer, hat aber eine weitere Funktionalität, die jedoch erst beim Thema Vererbung relevant wird. Bis dahin werden nur die beiden anderen Modifizierer betrachtet.

Syntax in Java:

Optional kann die Klasse mit dem `public`-Modifizierer versehen werden. Damit steht sie auch anderen Paketen zur Verfügung. Wird die Klasse ohne `public`-Modifizierer angegeben, dann ist sie nur in dem eigenen Paket verfügbar.

```
[public] class  Name {

    [ Attribute ]
    [ Methoden ]
}
```

Beliebig viele Attribute können angelegt werden.

Beliebig viele Methoden können angelegt werden. Zusätzlich gibt es noch spezielle Methoden (die Konstruktoren und den Destruktor, dazu später mehr).

Erstes Beispiel einer Klasse

```
package kapitel_6;


class ErsteKlasse {

        public int x = 10;
        private String s = "Hallo";
        int ohneModifizierer = 10;


}
```

In einer Datei kann es immer nur **eine „Hauptklasse" geben, die so wie die Datei heißt**. Weitere Klassen (so wie die Klasse `ErsteKlasse`) dürfen deshalb nicht `public` sein.

Die Klasse `ErsteKlasse` wird definiert. In der Klasse sind drei Attribute vorhanden. Ein Attribut ist „`public`", ein Attribut ist „`private`" und das dritte Attribut hat keinen Modifizierer.

```
public class Hauptklasse {
    public static void main(String[] args) {

            ErsteKlasse objektVerweis;
            objektVerweis= new ErsteKlasse();

            objektVerweis.x = 20;

            objektVerweis.s = "Neu";

            objektVerweis.ohneModifizierer = 10;

    }
}
```

Ein Objekt der Klasse wird angelegt.

Der Zugriff auf das public-Attribut funktioniert.

Dieser Zugriff ist verboten, weil s ein privates Attribut ist!

Der Zugriff auf das Attribut ohne Modifizierer funktioniert auch!

In diesem ersten Beispiel sind einige neue Aspekte zu klären:

- Eine neue Klasse wird definiert, aber nicht innerhalb der „Hauptklasse" (das kann auch sinnvoll sein, später dazu mehr). Der Name der neuen Klasse ist frei wählbar (siehe Konventionen für Variablennamen).
- Das Erstellen eines Objektes der neuen Klasse geschieht ähnlich wie das Anlegen einer Variablen von einem elementaren Datentyp. Statt des Datentyps wird aber der Klassenname verwendet. Die Klasse ist im Prinzip ein neu geschaffener (benutzerdefinierter) Datentyp. In einem ersten Schritt wird ein sogenannter Verweis auf die Klasse angelegt:

```
ErsteKlasse objektVerweis;
```

Verweis anlegen

- Anschließend kann diesem Verweis dann ein konkretes Objekt im Speicher zugeordnet werden. Mit dem new-Operator wird ein solches Objekt im Speicher angelegt und dem Verweis zugewiesen:

```
objektVerweis = new ErsteKlasse();
```

Objekt im Speicher mit new anlegen und dem Verweis zuordnen.

- Der Zugriff auf ein Attribut des Objektes geschieht durch den Punktoperator.

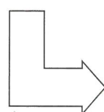
```
objektVerweis.x = 20;
```

Punktoperator

Public-Attribute können direkt angesprochen werden. Allerdings widersprechen public-Attribute einem Grundprinzip der OOP – siehe auch nächste Erläuterung.

- Private-Attribute können nicht von außen angesprochen werden. Nach dem Starten des Programms erscheint der folgende Compilerfehler:

```
Fehler:
s has private access
```



```
objektVerweis.s = "Neu";
```

Fehlerzeile

Dieser Fehler macht darauf aufmerksam, dass versucht wurde, auf ein privates Attribut zuzugreifen. Das wird vom Compiler verhindert, denn private Atrribute sollen nicht von außen zugänglich sein. Das entspricht einem Grundprinzip der objektorientierten Programmierung – der **Kapselung**. Nun kann es natürlich nicht die Lösung sein, alle Attribute mit dem `public`-Modifizierer zu versehen, denn damit würde gegen dieses Grundprinzip verstoßen. Vielmehr müssen andere geeignete Mechanismen entwickelt werden, um kontrolliert auf die Attribute zugreifen zu können. Mithilfe der Methoden im nachfolgenden Unterkapitel kann dieses Problem gelöst werden.

Hinweis

Attribute, die ohne Modifizierer angegeben werden (wie das Attribut `ohneModifizierer` in dem obigen Beispiel), haben im Prinzip das Verhalten eines privaten Attributes. Für Klassen innerhalb eines Paketes stellt sich der Zugriff allerdings so dar, als wäre das Attribut öffentlich (`public`) deklariert.

6.1.2 Werttypen und Verweistypen

Bislang wurden Variablen von elementaren Datentypen (Werttypen) einfach angelegt und konnten benutzt werden. Dies lag daran, dass diese Variablen in einem bestimmten Speicherbereich abgelegt wurden – dem **STACK**-Speicher[1]. Mit der Einführung der Klassen in Java kommt ein neuer Typ ins Spiel, und zwar der Verweistyp. Alle Objekte, die von Klassen gebildet werden, werden in einem anderen Speicherbereich abgelegt – dem **HEAP**-Speicher. Damit auf das Objekt zugegriffen werden kann, muss nun ein Verweis auf das Objekt angelegt werden. Das kann in zwei Schritten (siehe obiges Beispiel) oder auch in einem Schritt geschehen, wie das folgende Beispiel zeigt:

```
ErsteKlasse objektVerweis = new ErsteKlasse();
```

Verweis der Klasse `ErsteKlasse`

Mit dem `new`-Operator ein Objekt dynamisch im HEAP-Speicher anlegen und dem Verweis zuordnen

Der garbage collector

Alle Werttyp-Variablen werden auf dem STACK gespeichert und auch wieder automatisch gelöscht, wenn sie ihre Gültigkeit verlieren. Im Gegensatz dazu wird ein Objekt auf dem HEAP gespeichert und erst dann gelöscht, wenn kein Verweis mehr für dieses Objekt existiert, denn es können durchaus mehrere Verweise auf dasselbe Objekt existieren. Dieses Löschen wird durch den sogenannten **garbage collector** durchgeführt. Dieser Mechanismus erkennt „verweislose" Objekte und entfernt sie aus dem Speicher. In anderen Programmiersprachen wie beispielsweise C++ musste dieses Löschen vom Programmierer selbst durchgeführt werden, wodurch eine erhebliche Fehlerquelle entstand.

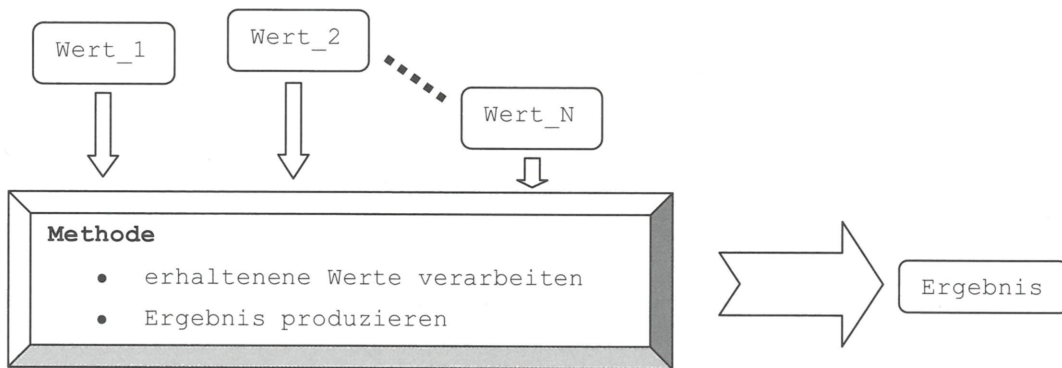
6.2 Methoden in Java

Aus den ersten Beispielen einer Klasse wurde deutlich, dass der Zugriff auf die Attribute über einen Mechanismus erfolgen muss, der auch zusätzlich Kontrollmöglichkeiten bietet. Beispielsweise wäre es nicht sinnvoll, dem Attribut *PS* eines Rennwagens einen negativen Wert zuzuweisen. An dieser Stelle müsste eine Methode diese unsinnige Zuweisung verhindern.

6.2.1 Aufbau einer Methode

Eine Methode ist technisch gesehen nichts anderes als eine Funktion. Sie wird aufgerufen und erfüllt eine bestimmte Aufgabe. Anschaulich kann man sich eine Methode wie einen Apparat vorstellen, der Eingaben (Werte) erhält und ein Ergebnis produziert.

1 Der STACK-Speicher ist ein bestimmter Speicherbereich, der für lokale (begrenzt gültige) Variablen genutzt wird. Er arbeitet nach dem LIFO-Prinzip (*Last in first out*). Der HEAP-Speicher ist hingegen ein Bereich, in dem Platz für Objekte bereitgestellt wird.



In allen bisherigen Beispielen wurden bereits Methoden (intuitiv) verwendet. Die wichtigste dabei war die statische `main`-Methode. Diese Methode wird beim Starten des Programms aufgerufen und ausgeführt. Daran sieht man, dass innerhalb einer Methode genauso programmiert wird, wie es in den vorherigen Kapiteln der Fall war.

Die wichtigsten Eigenschaften von Methoden im Überblick:

- Methoden haben einen Bezeichner (Namen), der wie bei den Variablen gebildet wird. Nach dem Bezeichner steht immer ein rundes Klammerpaar (entweder leer oder mit Parametern versehen).
- Methoden haben einen sogenannten Rumpf, in dem die Methode programmiert wird. Der Rumpf wird in geschweiften Klammern eingefasst.
- Methoden können beliebig viele Werte (Parameter) übernehmen.
- Methoden können einen Wert zurückgeben.

Das erste einfache Beispiel einer Methode

```
class Person {

      private String name;

      public void initName() {

         name = "Kaiser";
      }

}

public class Hauptklasse {
    public static void main(String[] args) {

           Person einePerson = new Person();

           einePerson.initName();
    }
}
```

Die Klasse `Person` soll für eine beliebige Person stehen. Der Einfachheit halber wird zuerst nur ein Attribut (`name`) angelegt.

Die Methode `initName()` initialisiert das private Attribut `name` der Klasse `Person`.

Ein Objekt der Klasse wird angelegt.

Die Methode `initName()` wird mithilfe des Punktoperators aufgerufen.

An dem Beispiel ist ersichtlich, dass die Methode `initName()` von einem Objekt der Klasse `Person` aufgerufen werden kann. Dies liegt daran, dass die Methode „`public`" ist. Die Methode selbst ist vom Typ **`void`**. Das bedeutet, dass die Methode keinen Wert an die aufrufende Stelle zurückgibt – der Datentyp `void` steht also für **keine Rückgabe** (dazu später mehr). Die Methode hat weiterhin ein leeres rundes Klammerpaar. Dadurch übernimmt die Methode keine Werte (auch dazu später mehr). In dem obigen Beispiel handelt es sich also um die einfachste Form einer Methode.

Im nächsten Beispiel wird eine weitere Methode ergänzt, die den Namen der Person auf den Bildschirm schreibt:

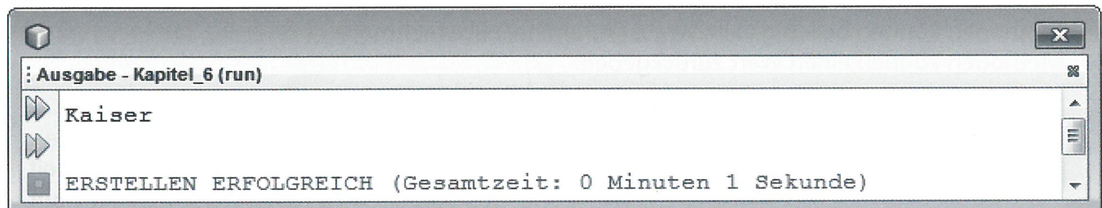
```
class Person {
        :
        :
        public void schreibeName() {

                System.out.println(name);
                System.out.println();
        }
}
```

Die Methode `schreibeName()` schreibt den Namen der Person auf den Bildschirm.

```
public class Hauptklasse {
        public static void main(String[] args) {

                Person einePerson = new Person();
                einePerson.initName();
                einePerson.schreibeName();
        }
}
```

Aufruf der Methoden durch Angabe des Namens und der leeren Klammern

Nach dem Starten sieht die Bildschirmausgabe so aus:



6.2.2 Rückgabewert einer Methode

Die Methode `schreibeName()` aus dem obigen Beispiel schreibt den Namen einer Person auf den Bildschirm. Nun soll aber der Name der Person einer anderen `String`-Variablen zugewiesen werden. Dazu müsste eine Methode den Namen zurückgeben können. Das kann durch folgende Anpassung geschehen:

Rückgabedatentyp der Methode

```
public String gibName() {

        return name;
}
```

Rückgabe eines Wertes mit `return`

Mithilfe dieser Methode kann der Name einer `String`-Variablen zugewiesen werden:

```
public class Hauptklasse {
  public static void main(String[] args) {


        Person einePerson = new Person();

        String einName;

        einePerson.initName();
        einName = einePerson.gibName();
        System.out.println(einName);
  }
}
```

Die Methode gibt den Namen zurück.

Die obige Zuweisung funktioniert deshalb, weil die Methode nach ihrem Aufruf einen Wert zurückgibt und dieser Wert dann anstelle des Methodenaufrufes steht:

```
einName = einePerson.gibName();

einName =       "Kaiser";
```

Nach dem Aufruf

Allgemein kann der Aufbau einer Methode mit Rückgabewert so geschrieben werden:

```
Modifizierer Rückgabedatentyp Bezeichner ()
{
    Anweisung_1;
    Anweisung_2;
    :
    Anweisung_N;

    return wert;
}
```

Der Rückgabedatentyp und der Datentyp des Rückgabewertes müssen übereinstimmen.

Das folgende Beispiel zeigt eine Methode, die einen Rückgabedatentyp `double` hat, aber einen `String` zurückgibt. Das passt natürlich nicht zusammen.

```
public double schlechtesBeispiel() {
    String zurueck = "Hallo";
    return zurueck;
}
```

Compilerfehler:
`incompatible types`
`required: double`
`found: java.lang.String`

Hinweise:

- Bei der Rückgabe von Werttypen mit `return` wird eine Kopie des Rückgabewertes erstellt und an die aufrufende Stelle zurückgegeben.
- Bei der Rückgabe von Verweistypen mit `return` wird der Verweis zurückgegeben.

6.2.3 Lokale Variablen

Variablen, die in einer Methode angelegt werden, sind nur innerhalb dieser Methode gültig. Wird eine Methode aufgerufen, so werden diese Variablen verarbeitet und nach Beendigung „gelöscht" (sie haben also eine lokale Gültigkeit). Objekte können ebenfalls in einer Methode angelegt werden. Sie werden vom **garbage collector** gelöscht, sobald kein Verweis mehr auf sie existiert. Möchte man das „Überleben" eines Objektes sichern, so müsste der Verweis auf das Objekt zurückgegeben werden. Das folgende Beispiel demonstriert diese Problematik:

```
package kapitel_6;

class Person { ... }

class Test {

    public Person gibeinePerson() {

        Person einePerson = new Person();
        return einePerson;
    }

    public void lokaleVariablen() {

        int x = 10;
        double d = 1.25;
    }
}
```

Diese Methode erzeugt ein Objekt auf dem HEAP-Speicher mit einem lokalen Verweis. Der Verweis wird aber von der Methode zurückgegeben.

Diese Methode legt zwei lokale Variablen auf dem STACK-Speicher an. Nach dem Aufruf der Methode werden diese Variablen wieder gelöscht.

```
public class Hauptklasse {
      public static void main(String[] args) {

            Test einTest = new Test();
            Person neuePerson = einTest.gibeinePerson();

            neuePerson.initName();
            neuePerson.schreibeName();

            einTest.lokaleVariablen();
      }
}
```

Mithilfe der Methode `gibeinePerson()` wird eine Person erzeugt und dem Verweis `neuePerson` zugewiesen. Auch nach dem Aufruf der Methode ist das Personen-Objekt gültig, da ein Verweis darauf existiert.

Das Personenobjekt kann weiter benutzt werden.

Diese Methode wird aufgerufen und die zwei lokalen Variablen werden erzeugt. Nach dem Aufruf sind die lokalen Variablen gelöscht.

Hinweis

Alle Variablen, die bislang in der `main`-Methode angelegt wurden, sind selbstverständlich auch nur lokal gültig. Da die `main`-Methode aber im Prinzip das „Hauptprogramm" ist, behalten die Variablen während der gesamten Laufzeit ihre Gültigkeit.

6.2.4 Übergabeparameter einer Methode

Eine Methode kann nicht nur einen Wert zurückgeben, sondern auch Werte übernehmen. Das geschieht durch sogenannte Parameter, die in den runden (bislang leeren) Klammern einer Methode angegeben werden können. Mehrere Parameter werden durch Kommata getrennt.

Allgemein kann der Aufbau einer Methode mit Rückgabewert und Parametern so geschrieben werden:

```
Modifizierer Rückgabedatentyp Bezeichner (Typ param_1,Typ param_2,...) {

      Anweisung_1;

      Anweisung_2;
      :
      Anweisung_N;

      return wert;
}
```

In den Parametern (auch Übergabevariablen genannt) sind die Werte gespeichert, die der Methode übergeben werden. Jeder Parameter hat einen Datentyp und einen Namen. Die Parameter sind durch Kommata getrennt.

Das folgende Programm zeigt die Verwendung von Parametern:

```
class Person {

      private String name;
      private double gewicht;


      public void setzeName(String nameParam) {

            name = nameParam;
      }
```

Die Methode `setzeName()` der Klasse `Person` kann einen `String` übernehmen und dem Attribut `name` zuweisen.

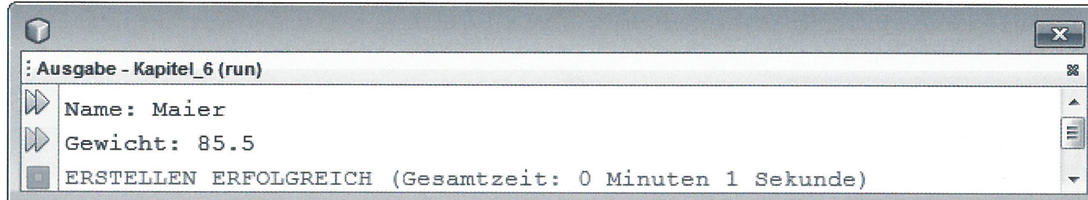
Die Methode `setzeAlleWerte()` der Klasse `Person` kann einen `String` und einen `double`-Wert übernehmen und den entsprechenden Attributen zuweisen.

```
    public void setzeAlleWerte(String nameParam,double gewichtParam){
        setzeName(nameParam);
        gewicht = gewichtParam;
    }

    public String gibName() {
        return name;
    }
    public double gibGewicht() {
        return gewicht;
    }
}
public class Hauptklasse {
    public static void main(String[] args) {
        Person einePerson = new Person();
        einePerson.setzeName("Kaiser");
        einePerson.setzeAlleWerte("Maier",85.5);
        System.out.println("Name: " + einePerson.gibName());
        System.out.println("Gewicht: " + einePerson.gibGewicht());
    }
}
```

Die vorhandene Methode `setzeName()` wird einfach genutzt. Methoden derselben Klasse dürfen natürlich in Methoden aufgerufen werden.

Die Methoden werden aufgerufen und Werte werden übergeben.

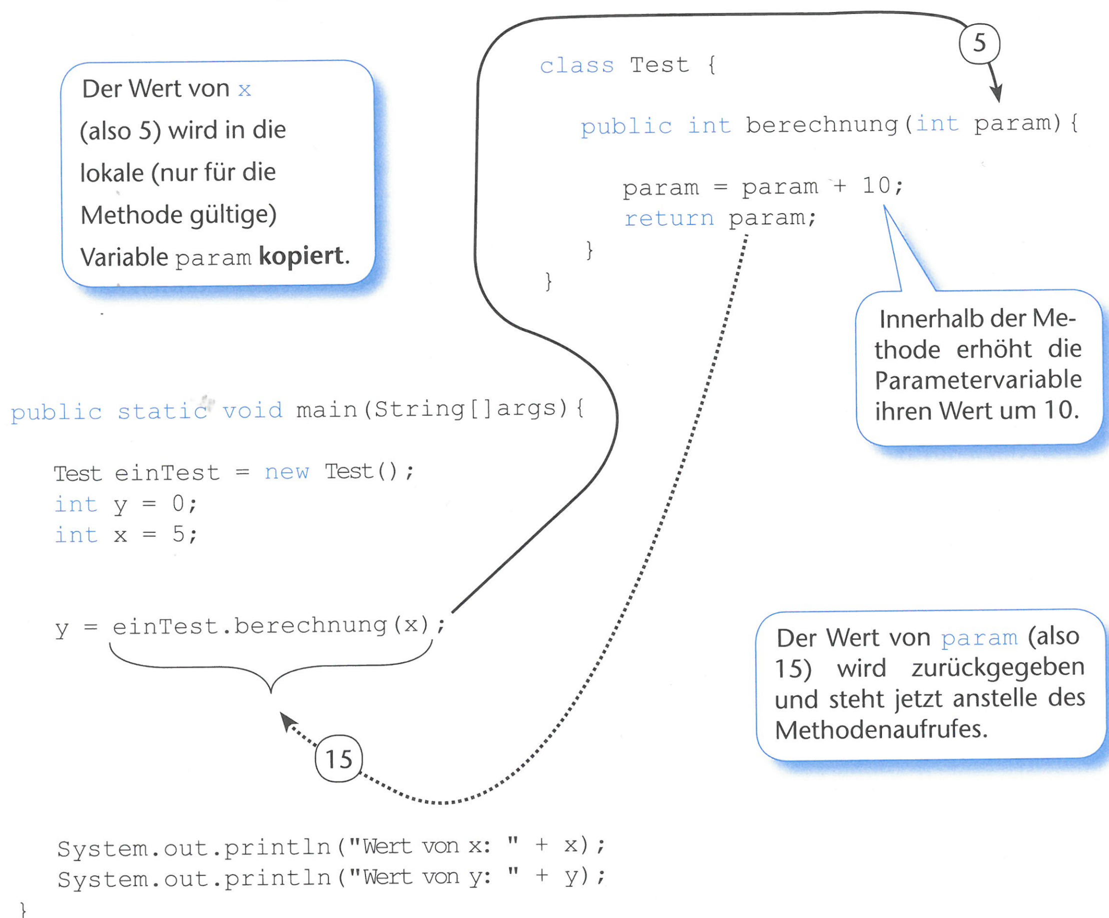
Nach dem Starten sieht die Bildschirmausgabe so aus:



Hinweis

Eine Methode kann beliebig viele Parameter haben. Die Parameter sind nichts anderes als lokale Variablen, in denen Werte gespeichert sind, die der Methode beim Aufruf übergeben werden. Mit den Parametern kann wie mit allen anderen lokalen Variablen einer Methode gearbeitet werden.

Das Verständnis für Übergabeparameter und Rückgabewerte einer Methode ist sehr wichtig, deshalb wird der Zusammenhang noch einmal grafisch verdeutlicht:



Nach dem Starten sieht die Ausgabe so aus:

```
Ausgabe - Kapitel_6 (run)
Wert von x: 5
Wert von y: 15
ERSTELLEN ERFOLGREICH (Gesamtzeit: 0 Minuten 1 Sekunde)
```

Die Variable y hat ihren Wert verändert. Sie hat den Rückgabewert der Methode erhalten. **Die Variable x hat ihren Wert nicht verändert**. Sie hat ihren Wert nur für eine Kopie zur Verfügung gestellt. Diese Art der Übergabe nennt man **call by value**. Im Gegensatz zu anderen Sprachen gibt es in Java nur diese Art der Übergabe – in C++ oder C# kann ein Parameter auch als sogenannte Referenz übergeben werden.

Übergabe von Verweistypparametern

Bei Verweistypen verhält sich die Parameterübergabe auf den ersten Blick etwas anders als bei den Werttypen. Wenn ein Verweistyp übergeben wird, dann wird der Verweis kopiert und damit verweisen sowohl der Parameter als auch der übergebene Verweis auf dasselbe Objekt im Speicher. Das folgende Beispiel zeigt den Unterschied zur Werttyp-Übergabe:

```
class Verweis {

    private int x;

    public int gibwert() {
        return x;
    }
```

> Die Klasse `Verweis` dient als einfaches Beispiel für die Übergabe eines Verweises. Es kann ein Integerwert gespeichert werden.

```
    public void setzeWert(int param) {
        x = param;
    }
}

class VerweisParameter {
```

Die Klasse `VerweisParameter` implementiert eine Methode, die einen Verweisparameter hat.

```
    public void uebergabe_1(Verweis param){

        param.setzeWert(10);

    }
```

Von dem übergebenen Verweis wird die Methode `setzeWert()` aufgerufen.

```
   public void uebergabe_2(Verweis param){

        param = new Verweis();
        param.setzeWert(30);
    }
}
```

Dem Verweisparameter wird dynamisch ein neues Objekt zugewiesen und anschließend die Methode `setzeWert()` aufgerufen.

```
public class Hauptklasse {
      public static void main(String[] args) {

        VerweisParameter einTest = new VerweisParameter();
        Verweis einVerweis = new Verweis();
```

Objekte der Klassen `VerweisParameter` und `Verweis` werden instanziert.

```
        einVerweis.setzeWert(0);
```

Das Objekt `einVerweis` erhält einen Wert (Null).

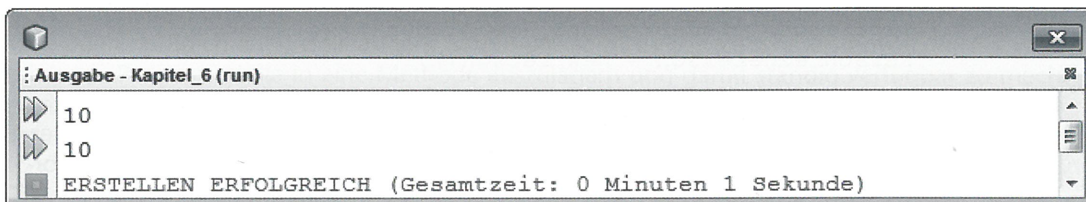
Übergabe an die Methode `uebergabe_1()`

```
        einTest.uebergabe_1(einVerweis);
        System.out.println(einVerweis.gibwert());
```

Übergabe an die Methode `uebergabe_2()`

```
        einTest.uebergabe_2(einVerweis);
        System.out.println(einVerweis.gibwert());
      }
}
```

Nach dem Starten sieht die Bildschirmausgabe so aus:



An den Aufrufen ist erkennbar, dass die Übergabe eines Verweises zwar Veränderungen an dem Objekt zulässt, aber keine neuen Objekte instanziert werden können, die dann auch dem übergebenen Verweis zugeordnet werden können.

Hinweis

Die Übergabe von Verweistypen geschieht zwar auch nach dem ***call by value*** – Prinzip, ermöglicht aber eine Veränderung des Objektes, dessen Verweis übergeben wurde. Damit ähnelt diese Übergabe dem sogenannten ***call by reference***, obwohl es kein echter Referenzaufruf ist.

6.2.5 Überladen von Methoden

Beim Überladen von Methoden geht es darum, dass Methoden denselben Namen haben und ähnliche Aufgaben erfüllen, allerdings für verschiedene Übergabeparameter. Die Überladung ist eine wichtige Eigenschaft, die vor allem bei den Konstruktoren (siehe nächstes Kapitel) eingesetzt wird.

Beispiel:

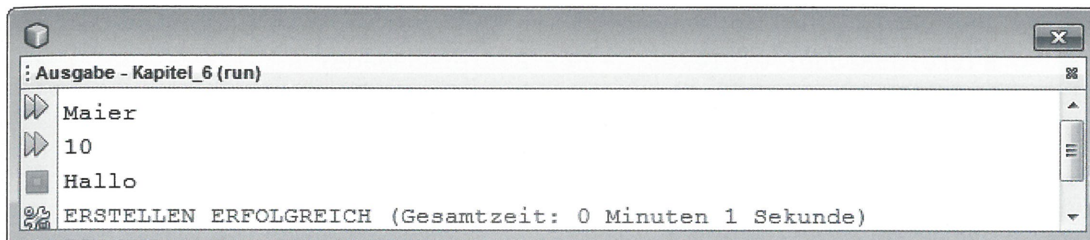
Es sollen Methoden geschrieben werden, die den Inhalt eines Übergabeparameters auf dem Bildschirm ausgeben. Für verschiedenen Datentypen wird eine eigene Methode implementiert.

```
class Person {

         private String name = "Maier";

         public String gibName() {
                return name;
         }

}

class Ueberladen {

  public void ausgabe(Person personParam) {

        System.out.println(personParam.gibName());

  }

  public void ausgabe(int intParam) {

        System.out.println(intParam);
  }

  public void ausgabe(String stringParam) {

        System.out.println(stringParam);
  }
}


public class Hauptklasse {
  public static void main(String[] args) {

        Ueberladen einTest = new Ueberladen();
        Person einePerson = new Person();

        einTest.ausgabe(einePerson);
        einTest.ausgabe(10);
        einTest.ausgabe("Hallo");
    }
 }
```

Drei Methoden mit gleichem Bezeichner, aber unterschiedlichen Parametertypen

Je nach Übergabeparameter erkennt der Compiler die korrekte Methode.

Nach dem Starten sieht die Bildschirmausgabe so aus:



Vorteil:

Der Programmierer kann ähnliche Aufgaben mit demselben (Methoden-)Namen benennen – dadurch wird das Programmieren einfacher und übersichtlicher.

Hinweis:

Dem Überladen von Methoden sind keine Grenzen gesetzt – allerdings muss der Compiler immer eindeutig unterscheiden können, welche der Methoden aufgerufen werden soll. Die folgenden Methoden sind keine korrekt überladenen Methoden:

```
public int methode() { return 10; }
public double methode() { return 10.5; }
```

Eine Unterscheidung durch den Rückgabedatentyp **reicht nicht aus.**

6.2.6 Zusammenfassende Hinweise zu Methoden

Die bisherigen Ausführungen zu Methoden haben eher die technische Seite (Aufbau, Rückgabewert und Parameter) beleuchtet. Die nachfolgenden Anmerkungen sollen diese Ausführungen ergänzen und den Eindruck über die Möglichkeiten der Methoden vervollständigen.

Überprüfung der Attributzuweisungen

Eine wichtige Funktion der Methoden ist das Setzen und Zurückgeben von Attributwerten. Wenn Attribute nur über Methoden einen Wert erhalten, so kann sichergestellt werden, dass keine unsinnigen Attributwerte entstehen. Das kann enorm wichtig sein, denn unsinnige Attributwerte können ein ganzes Programm abstürzen lassen.

Beispiel:

Das Attribut `ps` eines Rennwagenobjektes wird auf Sinnhaftigkeit geprüft. Nur wenn der Übergabeparameter in bestimmten Grenzen liegt, wird das Attribut neu gesetzt.

```
public void setzeps(int psParam) {

      if (psParam < 1 || psParam > 3500) ps = 1;
      else ps = psParam;
}
```

Wiederkehrende Aufgaben in Methoden auslagern

Neben der Funktionalität als sogenannte Get- und Set-Methoden wie im obigen Beispiel erfüllen Methoden natürlich auch viele andere Aufgaben. So ist es beispielsweise sinnvoll, immer wiederkehrende Programmteile in eine Methode auszulagern und damit ständig verfügbar zu machen. Wenn diese Programmteile nicht öffentlich, sondern nur innerhalb der Klasse verfügbar sein sollen, dann würde sich eine private Methode anbieten.

Beispiel:

Bei vielen Methoden einer Klasse soll so lange ein Wert über die Tastatur eingelesen werden, bis die Eingabe in bestimmten Grenzen und damit korrekt ist. Diese Eingaberoutine kann nun in eine private Methode ausgelagert und von anderen Methoden genutzt werden.

```
private int einlesen()  throws IOException {

   int eingabe;
   BufferedReader einlesen  = new BufferedReader
                              (new InputStreamReader(System.in));

      do {

         System.out.println("Bitte einen Wert eingeben (>=0)");
         eingabe = Integer.parseInt(einlesen.readLine());


      }
      while (eingabe < 0);

      return eingabe;
}
public void andereMethode() throws IOException {
  int x = einlesen();
  int y = einlesen();
  int z = einlesen();
}
```

Eine immer wiederkehrende Aufgabe wird in eine private Methode ausgelagert.

Eine andere Methode nutzt die vorhandene `einlesen`-Methode.

Hinweis:

Jede Methode einer Klasse kann von jeder (***nicht statischen***) Methode derselben Klasse aufgerufen werden. Innerhalb der Klasse spielt es auch keine Rolle, ob die Methode `private`, `protected` oder `public` ist. Außerhalb der Klasse spielt es natürlich eine Rolle.

6.3 Weitere Elemente von Klassen

6.3.1 Konstruktoren und der Destruktor

Konstruktoren

Die Konstruktoren sind ganz spezielle Methoden einer Klasse, die von außen nicht aufrufbar sind, sondern implizit bei der Instanziierung von Objekten aufgerufen werden. Konstruktoren sind also bei der „Konstruktion" eines Objektes wichtig. Sie übernehmen in der Regel initialisierende Aufgaben. Das können Zuweisungen an Attribute oder auch das Herstellen einer Datenbankverbindung oder das Öffnen einer Datei sein (dazu später mehr). Konstruktoren haben folgende Eigenschaften:

- Konstruktoren heißen so wie der Klassenname.
- Konstruktoren können nicht explizt aufgerufen werden.
- Konstruktoren haben keinen Rückgabedatentyp und damit auch keinen Rückgabewert.
- Sie können beliebig oft überladen werden.
- Ein Konstruktor ohne Parameter heißt **Standardkonstruktor**.
- Ein Konstruktor mit Parametern heißt **Parameterkonstruktor**.

Das folgende Programm zeigt verschiedene Konstruktoren und deren Aufruf bei der Instanziierung von Objekten.

```
class Kontakt {


   private String name;
   private String telefon;


   public Kontakt() {
       name = "LEER";
       telefon = "LEER";
   }
```

Die Klasse `Kontakt` soll einen einfachen Kontakt mit Name und Telefon repräsentieren.

Der Standardkonstruktor initialisiert die Attribute mit der Zeichenkette „`LEER`".