Klassen und Objekte

- Deklaration von Klassen
- Objekte
- Konstruktoren
- Überladen von Methoden
- Parameterübergabe-Mechanismen
- Das Schlüsselwort `this`
- Klassen- und Objektattribute
- Klassenmethoden
- Klassenbezogene Initialisierungsblöcke
- Finalisierung eines Objekts
- Die Methode `toString()`

Klassen und Objekte

Die Begriffe Klasse und Objekt

Eine **Klasse** ist eine Schablone, welche die *Attribute* und *Operationen* beschreibt, die jedes Objekt dieser Klasse besitzt.

Attribute sind die Datenkomponenten, die ein Objekt der Klasse enthält.

Operationen beschreiben die Aktionen, die auf den Attributen der Objekte ausgeführt werden können.

Operationen werden in Java **Methoden** genannt.

Ein **Objekt** wird als **Exemplar** oder **Instanz** einer Klasse bezeichnet.

Klassen und Objekte

Beispiel: Die Klasse Studentenkonto

```
// Beschreibung der Klasse StudentenKonto

// Ein StudentenKonto verwaltet die Konteninformationen
// für einen Studenten.

// Attribute:
    String matrikelnr;          // Matrikelnr. in der Form ##-####
    String name;                // Name des Studenten
    double kontostand;          // aktueller Kontostand

// Operationen:
    // Konstruktoren
    StudentenKonto() {}
    StudentenKonto(String mnr, String name, double betrag) {}

    // Buchung durchführen; Kontostand aktualisieren
    void belastung(double betrag) {}
    void gutschrift(double betrag) {}

    // aktuelle Werte der Attribute zurückliefern
    String getMatrikelnr() {}
    String getName() {}
    double getKontostand() {}

    // Kontoauszug drucken
    void druckeKontoauszug() {}
```

Klassen und Objekte

Deklaration von Klassen

Eine **Klasse** hat folgenden (vereinfachten) Aufbau:

Modifizierer-Liste
`class` ***Bezeichner Klassenrumpf***

Der **Klassenrumpf** hat die folgende Syntax:

{
Attributdeklarationen
Methodendeklarationen
Konstruktordeklarationen
statische Initialisierer
Klassendeklarationen
}

Die **Attributdeklaration** hat die gleiche Syntax wie die Variablendeklaration:

Modifizierer-Liste Typname Bezeichner-Liste ;

Klassen und Objekte

Deklaration von Klassen

Beispiel: Deklaration der Klasse `StudentenKonto`

```
class StudentenKonto
{
  // Attribute:

  // Matrikelnr. in der Form ##-####
  String matrikelnr;

  // Name des Studenten
  String name;

  // aktueller Kontostand
  double kontostand;

  // Methoden:
  // ...
}
```

Deklaration von Klassen

Die **Methodendeklaration** hat folgenden Aufbau:

Methodenkopf Block

Der **Methodenkopf** hat folgende (vereinfachte) Syntax:

Modifizierer-Liste Typname
Bezeichner **(** ***formale-Parameter-Liste*** **)**

Die **formale-Parameter-Liste** kann entweder leer sein oder aber eine oder mehrere **Parameterdeklarationen** enthalten, die durch Komma getrennt sind:

Typname Bezeichner

Klassen und Objekte

Deklaration von Klassen

Beispiel: Deklaration der Klasse `StudentenKonto`

```
class StudentenKonto
{
  // Attribute:
  String matrikelnr, name; double kontostand;

  // Methoden:

  void belastung(double betrag)
  {
    if (betrag < 0)
    {
      System.out.println(
        "Der Betrag muss >= 0 sein!");
      System.exit(1);
    }
    kontostand -= betrag;
  }

  String getName()
  {
    return name;
  }

  void druckeKontoauszug()
  {
    System.out.println("Matrikelnr.: " + matrikelnr);
    System.out.println("Name:        " + name);
    System.out.println("Kontostand:  " +
      kontostand + " DM");
  }
}
```

Klassen und Objekte

Objekte

Objekte deklarieren

In Java *deklariert* man ein Objekt (eigentlich: eine *Referenz auf ein Objekt*) durch Angabe des Klassentyps gefolgt von einem Bezeichner.

Beispiel:

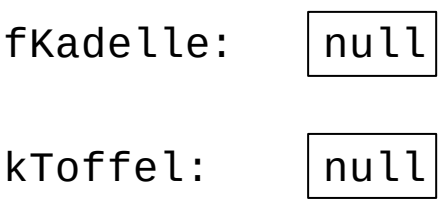
```
StudentenKonto fKadelle, kToffel;
```

Diese Objekte stellen nicht den Speicherplatz für das eigentliche Exemplar der Klasse `StudentenKonto` zur Verfügung: sie können lediglich eine *Referenz* auf ein Objekt der Klasse `StudentenKonto` aufnehmen. Die Referenzen werden mit dem Wert `null` initialisiert.

Nach der Deklaration

```
StudentenKonto fKadelle, kToffel;
```

liegt also folgende Situation vor:



Klassen und Objekte

Objekte

Objekte erzeugen

In Java *erzeugt* man ein Objekt mit dem Operator new.

Die **Objekterzeugung** hat folgende Syntax:

new *Klassenname* (*Ausdrucks-Liste*)

Die **Ausdrucks-Liste** kann entweder leer sein oder aber einen oder mehrere **Ausdrücke** enthalten, die durch Komma getrennt sind.

Die Objekterzeugung reserviert dynamisch (d. h. während des Programmlaufs) den von einem Objekt der Klasse benötigten Speicherplatz und liefert eine Referenz auf das neue Objekt.

Beispiel:

```
fKadelle = new StudentenKonto();
kToffel  = new StudentenKonto("67-5634",
             "Karl Toffel", 3600.0);
```

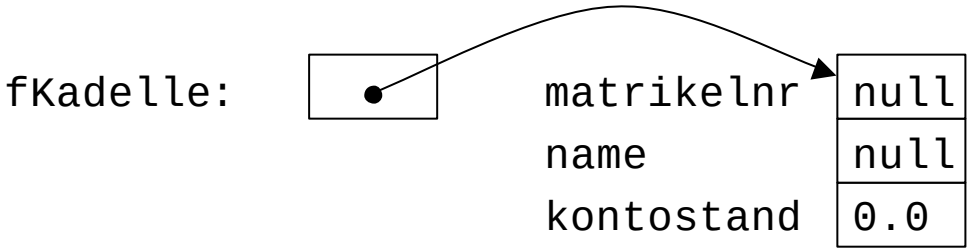
Objekte

Objekte erzeugen

Die Objekterzeugung

```
fKadelle = new StudentenKonto();
```

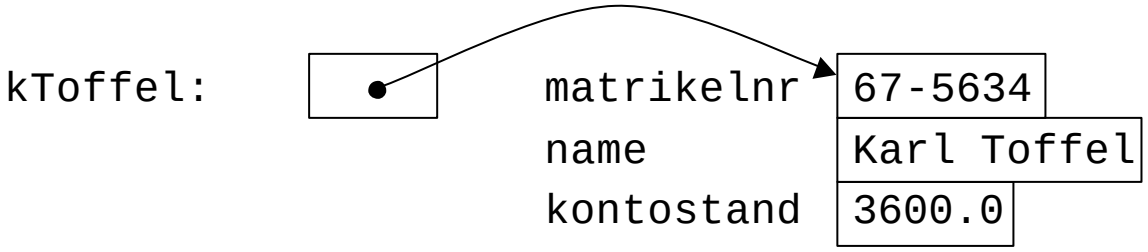
hat folgende Wirkung:



Die Objekterzeugung

```
kToffel  = new StudentenKonto("67-5634",
             "Karl Toffel", 3600.0);
```

hat folgende Wirkung:



Klassen und Objekte

Objekte

Objekte verwenden

Ein **Attributzugriff** hat die folgende (vereinfachte) Syntax:

Objektname **.** ***Attributname***

Ein **Methodenaufruf** hat die folgende (vereinfachte) Syntax:

Objektname **.** ***Methodennname*** **(** ***Ausdrucks-Liste*** **)**

Die **Ausdrucks-Liste** kann entweder leer sein oder aber einen oder mehrere **Ausdrücke** enthalten, die durch Komma getrennt sind.

Beispiel:

```
fKadelle.matrikelnr = "23-9876";
fKadelle.name = "Fritz Kadelle";
fKadelle.kontostand = 0.0;
System.out.println(fKadelle.kontostand);

fKadelle.gutschrift(1000.0);
fKadelle.belastung(39.95);
System.out.println(fKadelle.getKontostand());
fKadelle.druckeKontoauszug();
```

Klassen und Objekte

Objekte

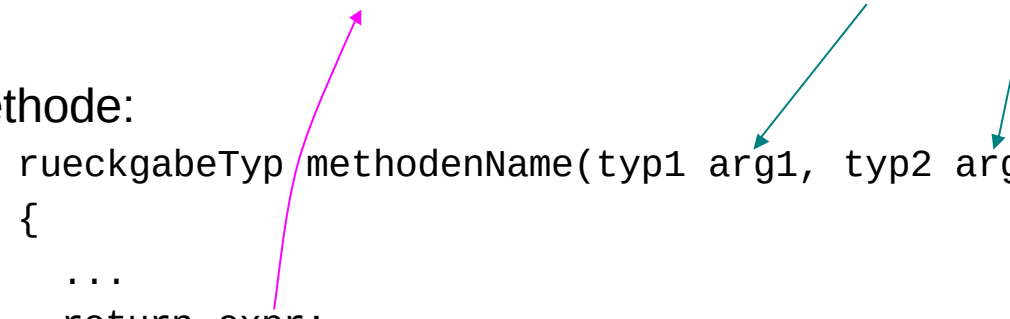
Methodenaufruf

Ein Methodenaufruf geschieht in drei Schritten:

- Parameterübergabe
- Ausführen des Methodenrumpfs
- Rückgabe des Funktionswerts

Methodenaufruf:

```
rueckgabeWert = objekt.methodenName(expr1, expr2, ...)
```



Methode:

```
rueckgabeTyp methodenName(typ1 arg1, typ2 arg2, ...)
{
  ...
  return expr;
}
```

Beispiel:

```
String getName()
{
  return name;
}

System.out.println(fKadelle.getName());
```

Klassen und Objekte

Objekte

Methodenaufruf

Beispiel:

```
void belastung(double betrag)
{
  if (betrag < 0)
  {
    System.out.println(
      "Der Betrag muss >= 0 sein!");
    System.exit(1);
  }
  kontostand -= betrag;
}

double getKontostand()
{
  return kontostand;
}


fKadelle.belastung(39.95);

fKadelle.belastung(fKadelle.getKontostand());

fKadelle.getKontostand();
```

Objekte

Kriterien für den Aufruf einer Methode

- Die Anzahl der aktuellen Parameter im Methodenaufruf muss mit der Anzahl der formalen Parameter in der Methodendeklaration übereinstimmen.
    - Die Methode `gutschrift()` muss also mit *einem* aktuellen Parameter aufgerufen werden

- Auf jeder Position muss der aktuelle Parameter einen Datentyp liefern, der zum Datentyp des entsprechenden formalen Parameters passt.

- Ein Methodenaufruf darf nur an einer solchen Stelle auftreten, an der auch eine Variable des Ergebnistyps der Methode stehen kann. Der Ergebniswert des Methodenaufrufs kann aber auch ignoriert werden.
    - Die Methode `getKontostand()` liefert ein Ergebnis vom Typ `double`, das z. B. ausgegeben - oder ignoriert - werden kann

- Ist der Methodentyp `void`, so liefert der Methodenaufruf keinen Ergebniswert. Der Methodenaufruf bildet dann eine Anweisung.

Objekte

Aufruf einer Methode

- Die Ausdrücke der aktuellen Parameterliste des Aufrufs werden ausgewertet.

- Die formalen Parameter werden mit den Ergebnissen der Ausdrücke initialisiert. Gegebenenfalls werden die Konvertierungsregeln angewendet, wie sie bei der Wertzuweisung definiert sind.

- Es wird an die erste Anweisung des Methoden-rumpfes verzweigt. Die Anweisungen werden wie gewohnt abgearbeitet, bis man auf eine `return`-Anweisung oder das Ende des Methodenrumpfes stößt.

- Hat die Methode einen von `void` verschiedenen Ergebnistyp, so muss sie eine `return`-Anweisung enthalten, der ein Ausdruck folgt; dieser wird ausgewertet und muss in den Ergebnistyp der Methode gewandelt werden können.

- Die Programmabarbeitung wird dann hinter der Aufrufstelle der Methode fortgesetzt.

Klassen und Objekte

Beispiel: Die Klasse `StudentenKonto`

```
class StudentenKonto
{
  // Attribute
  String matrikelnr;          // Matrikelnr.
  String name;                // Name des Studenten
  double kontostand = 0.0;    // Kontostand

  // Methoden:
  // Konstruktoren

  StudentenKonto()
  {
/*
    matrikelnr = "<##-####>";
    name = "<unbekannt>";
    kontostand = 0.0;
*/
  }

  StudentenKonto(String mnr, String nm,
                 double betrag)
  {
    matrikelnr = mnr;
    name = nm;
    kontostand = betrag;
  }
```

Klassen und Objekte

Beispiel: Die Klasse `StudentenKonto` (Fortsetzung)

```
// Buchung durchführen; Kontostand aktualisieren
// Der Betrag muss jeweils >= Null sein.
// Falls nicht, wird das Programm abgebrochen;
// andernfalls wird der Kontostand entsprechend
// aktualisiert.

void belastung(double betrag)
{
  if (betrag < 0)
  {
    System.out.println(
      "Der Betrag muss >= 0 sein!");
    System.exit(1);
  }
  kontostand -= betrag;
}

void gutschrift(double betrag)
{
  if (betrag < 0)
  {
    System.out.println(
      "Der Betrag muss >= 0 sein!");
    System.exit(1);
  }
  kontostand += betrag;
}
```

Klassen und Objekte

Beispiel: Die Klasse `StudentenKonto` (Fortsetzung)

```
  // aktuelle Werte der Attribute zurückliefern
  String getMatrikelnr()
  {
    return matrikelnr;
  }

  String getName()
  {
    return name;
  }

  double getKontostand()
  {
    return kontostand;
  }

  // Kontoauszug drucken
  void druckeKontoauszug()
  {
    System.out.println("Matrikelnr.: " + matrikelnr);
    System.out.println("Name:        " + name);
    System.out.println("Kontostand:  " + kontostand
                       + " DM");
  }
}
```

Klassen und Objekte

Beispiel: Die Klasse `StudentenKontoTest`

```
class StudentenKontoTest
{
  public static void main(String [] args)
  {
    StudentenKonto fKadelle, kToffel;

    fKadelle = new StudentenKonto();
    kToffel = new StudentenKonto();

    fKadelle.druckeKontoauszug();
    kToffel.druckeKontoauszug();
    System.out.println();


    fKadelle = new StudentenKonto("23-9876",
                 "Fritz Kadelle", 0.0);
    kToffel  = new StudentenKonto("67-5634",
                 "Karl Toffel", 3600.0);

    fKadelle.druckeKontoauszug();
    kToffel.druckeKontoauszug();
    System.out.println();


    fKadelle.gutschrift(1000.0);
    System.out.println(fKadelle.getKontostand());
    System.out.println();

    kToffel.belastung(79.95);
    kToffel.druckeKontoauszug();

  }
}
```

Klassen und Objekte

Beispiel: Die Klasse `StudentenKontoTest`

Ausgabe des Programms:

```
Matrikelnr.: null
Name:        null
Kontostand:  0.0 DM
Matrikelnr.: null
Name:        null
Kontostand:  0.0 DM

Matrikelnr.: 23-9876
Name:        Fritz Kadelle
Kontostand:  0.0 DM
Matrikelnr.: 67-5634
Name:        Karl Toffel
Kontostand:  3600.0 DM

1000.0

Matrikelnr.: 67-5634
Name:        Karl Toffel
Kontostand:  3520.05 DM
```

Konstruktoren

Einem neu erzeugten Objekt wird ein Anfangszustand zugewiesen. Attribute können bei ihrer Deklaration mit einem Wert initialisiert werden, was manchmal ausreicht, um einen sinnvollen Anfangszustand sicherzustellen.

Oft ist aber mehr als nur eine einfache Attribut-initialisierung zur Erzeugung eines Anfangszustands nötig; der erzeugende Code muss vielleicht Initialisierungswerte liefern oder Operationen ausführen, die nicht als einfache Zuweisungen ausgedrückt werden können.

Um mehr als einfache Initialisierungen bewerkstelligen zu können, können Klassen **Konstruktoren** enthalten.

Konstruktoren sind keine Methoden, aber methoden-ähnlich. Ein Konstruktor trägt denselben Namen wie seine Klasse, hat keine, einen oder mehrere Parameter und keinen Rückgabetyp. Er wird bei der Erzeugung eines Objekts mit new automatisch aufgerufen.

Konstruktoren

Die **Konstruktordeklaration** hat folgenden Aufbau:

Konstruktordeklarator Konstruktorrumpf

Der **Konstruktordeklarator** hat folgende Syntax:

Modifizierer-Liste Bezeichner
(*formale-Parameter-Liste*)

Die **formale-Parameter-Liste** kann entweder leer sein oder aber eine oder mehrere **Parameterdeklarationen** enthalten, die durch Komma getrennt sind:

Typname Bezeichner

Der **Konstruktorrumpf** ist ein Block, der als erste Anweisung auch einen expliziten Konstruktoraufruf enthalten kann.

Konstruktoren

Beispiel: Die Klasse `StudentenKonto`

```
class StudentenKonto
{
  // Attribute

  String matrikelnr;          // Matrikelnr.
  String name;                // Name des Studenten
  double kontostand = 0.0;    // Kontostand

  // Konstruktoren

  StudentenKonto()
  {
    matrikelnr = "<unbekannt>";
    name = "<unbekannt>";
  }

  StudentenKonto(String mnr, String nm,
                 double betrag)
  {
    matrikelnr = mnr;
    name = nm;
    kontostand = betrag;
  }

  //  ...

}
```

Konstruktoren

Der Java-Compiler generiert für jede Klasse, für die *kein* Konstruktor deklariert ist, einen parameterlosen Konstruktor mit leerem Konstruktorrumpf. Dieser Konstruktor wird **Standardkonstruktor** (engl. *default constructor*) genannt.

Für die Klasse `StudentenKonto` entspricht der Standardkonstruktor dem folgenden explizit angegebenen parameterlosen Konstruktor:

```
StudentenKonto()
{
}
```

Klassen und Objekte

Beispiel: Die Klasse `Punkt`

```
class Punkt
{
   int x, y;
   String name;

// Konstruktor
   /**
   *  Konstruktor für Punkt
   *  @param a x-Koordinate
   *  @param b y-Koordinate
   *  @param n Name des Punktes
   */
   Punkt(int a, int b, String n)
   { x = a; y = b; name = n; }

// Methoden
   /**
   *  Berechnet die Länge als Abstand zum Ursprung.
   *  @return Länge
   */
   double laenge()
   { return Math.sqrt(x*x + y*y); }

   /**
   *  Ausgabe des Namens und der Koordinaten
   */
   void ausgeben()
   {
      System.out.println(
         name + "(" + x + "," + y + ")");
   }
}
```

Überladen von Methoden

Jede Methode besitzt eine **Signatur**, die ihren Namen sowie die Anzahl und Typen ihrer Parameter definiert.

Zwei Methoden können denselben Namen haben, wenn ihre Signaturen unterschiedliche Anzahlen oder Typen von Parametern haben.

Dieses Merkmal wird als **Überladen** bezeichnet, denn zum Methodennamen wurde (mehr als eine) Bedeutung "dazugeladen".

Wird eine Methode aufgerufen, vergleicht der Übersetzer die Anzahl und Typen der Parameterwerte mit den verfügbaren Signaturen, um die passende Methode zu finden.

Überladen von Methoden

Beispiel: Überladen der Methode `maximum( )`

```
class Ueberladen
{
   static int maximum(int a, int b)
   {
      return a>b ? a : b;
   }

   static int maximum(int a, int b, int c)
   {
      return maximum(maximum(a,b), c);
   }

   static double maximum(double a, double b)
   {
      return a>b ? a : b;
   }

   static String maximum(String a, String b)
   {
      // lexikographische Ordnung
      // a.compareTo(b) liefert einen Wert > 0,
      // wenn a hinter b kommt
      return (a.compareTo(b) > 0) ? a : b;
   }

   static Punkt maximum(Punkt p1, Punkt p2)
   {
      return (p1.laenge() > p2.laenge()) ? p1 : p2;
   }
```

Überladen von Methoden

Beispiel: Überladen der Methode `maximum( )`

```
    static void main(String[] argv)
    {
       int i = 12, j = 17, k = 13;
       double f = 321.345, g = 3.21245e2;
       String s1 = "Java", s2 = "Jawa";
       Punkt p1 = new Punkt(12, 3, "P1"),
             p2 = new Punkt(5, 5, "P2");

       System.out.println("Maximum von " + i +
          " und " + j + " ist " + maximum(i, j));
       System.out.println("Maximum von " + f +
          " und " + g + " ist " + maximum(f, g));
       System.out.println("Maximum von \"" + s1 +
          "\" und \"" + s2 + "\" ist \"" +
          maximum(s1, s2) + "\"");
       System.out.println("Maximum von " + i +
          ", " + j + " und " + k + " ist " +
          maximum(i, j, k));

       System.out.print("Maximum von ");
       p1.ausgeben();
       System.out.print(" und ");
       p2.ausgeben();
       System.out.print(" ist ");
       maximum(p1, p2).ausgeben();
       System.out.println();
    }
}
```

Überladen von Methoden

Beispiel: Überladen der Methode `maximum()`

Ausgabe des Programms:

```
Maximum von 12 und 17 ist 17
Maximum von 321.345 und 321.245 ist 321.345
Maximum von "Java" und "Jawa" ist "Jawa"
Maximum von 12, 17 und 13 ist 17
Maximum von P1(12,3)
 und P2(5,5)
 ist P1(12,3)
```

Es wird jeweils die Version von `maximum()` ausgeführt, die zur Anzahl und den Typen der aktuellen Parameter passt.

Überladen von Methoden

Im folgenden Beispiel ist die Auswahl der passenden Methode nicht ganz so offensichtlich.

Zur Berechnung des Maximums zweier `long`-Zahlen gibt es keine genau passende Methode.

Beispiel: Überladen mit Typanpassung

```
class Ueberladen2
{
  static int maximum(int a, int b)
  {
    System.out.println("maximum(int a, int b)");
    return a>b ? a : b;
  }

  static float maximum(float a, float b)
  {
    System.out.println("maximum(float a, float b)");
    return a>b ? a : b;
  }

  static double maximum(double a, double b)
  {
    System.out.println("maximum(double a, double b)");
    return a>b ? a : b;
  }
```

Überladen von Methoden

Beispiel: Überladen mit Typanpassung

```
  // Erweiterung von long nach float und double
  // möglich.
  // Die "billigere" Variante (nach float) wird
  // benutzt.
  public static void main(String[] argv)
  {
    long l1 = 12L, l2 = 21L;
    System.out.println("Maximum von " + l1 +
       " und " + l2 + " ist " + maximum(l1, l2));
  }
}
```

Ausgabe des Programms:

```
maximum(float a, float b)
Maximum von 12 und 21 ist 21.0
```

Wenn keine der vorhanden Methoden genau zum Aufruf passt, verwendet der Java-Compiler folgende zwei Regeln zur Auswahl einer Methode:

- Erweiterung der aktuellen Parameter "nach oben"
- Verwendung der Methode mit der "einfachsten" Erweiterung

Überladen von Methoden

Beispiel: Überladen, Mehrdeutigkeit beim Aufruf

```
class UeberladenMehrdeutig
{
  static long maximum(int a, long b) {
    System.out.println("\tmaximum(int a, long b)");
    return a>b ? a : b;
  }

  static long maximum(long a, int b) {
    System.out.println("\tmaximum(long a, int b)");
    return a>b ? a : b;
  }

  public static void main(String[] argv) {
    int i = 12, j = 13;
    System.out.println("Maximum von " + i +
      " und " + j + " ist " + maximum(i, j));
  }
}
```

Der Java-Compiler stellt eine Mehrdeutigkeit fest und meldet:

```
UeberladenMehrdeutig.java:45:
Reference to maximum is ambiguous. It is defined in
  long maximum(long, int) and
  long maximum(int, long).
      " und " + j + " ist " + maximum(i, j));
                                  ^
1 error
```

Überladen von Methoden

Beispiel: Konstruktor überladen

```
class Punkt1
{
   int x, y;
   String name;

   Punkt1(int a, int b, String n)
   { x = a; y = b; name = n; }

   Punkt1()
   { x = 0; y = 0; name = "unbenannt"; }

   Punkt1(String n)
   { x = 0; y = 0; name = n; }

   Punkt1(int a, int b)
   { x = a; y = b; name = "unbenannt"; }

   double laenge()
   { return Math.sqrt(x*x + y*y); }

   void ausgeben()
   {
      System.out.println(
         name + "(" + x + "," + y + ")");
   }
}
```

Überladen von Methoden

Beispiel: Konstruktor überladen

```
class Punkt1Test
{
   public static void main(String[] argv)
   {
      Punkt1 p1 = new Punkt1(0, 2, "p1"),
             p2 = new Punkt1("p2"),
             p3 = new Punkt1(3, 2),
             p4 = new Punkt1();

      p1.ausgeben();
      p2.ausgeben();
      p3.ausgeben();
      p4.ausgeben();
   }
}
```

Ausgabe des Programms:

```
p1(0,2)
p2(0,0)
unbenannt(3,2)
unbenannt(0,0)
```

Überladen von Methoden

In Java gibt es keine Methoden mit Vorgabeparametern (engl. *default parameter*). Stattdessen kann man die Methoden geeignet überladen.

Beispiel: Methoden mit "Vorgabeparametern"

```
class VorgabeParameter {
  public static void main(String[] argv) {
    System.out.println("Aufruf Diagonale(5, 3)"
      + " liefert " + Diagonale(5, 3));
    System.out.println("Aufruf Diagonale(4) "
      + " liefert " + Diagonale(4));
    System.out.println("Aufruf Diagonale() "
      + " liefert " + Diagonale());
  }

  static double Diagonale(double laenge,
                          double breite)
  { return Math.sqrt(
      laenge*laenge + breite*breite); }

  //  Methode Diagonale mit einem Parameter.
  //  Breite ist 1.
  static double Diagonale(double laenge)
  { return Diagonale(laenge, 1); }

  // Methode Diagonale ohne Parameter.
  // Länge und Breite jeweils 1.
  static double Diagonale()
  { return Diagonale(1, 1); }
}
```

Parameterübergabe-Mechanismen

Im Rumpf einer Methode wird stets mit dem *Wert* des aktuellen Parameters gearbeitet. Diese Form der Parameterübergabe nennt man **Wertübergabe** (engl. *call by value*). Der Wert des aktuellen Parameters kann nicht geändert werden.

Beispiel: Methode `tausche()` - falscher Ansatz

```
class Tausche_Falsch {
  static void tausche(long a, long b) {
    long hilf;
    System.out.println("tausche:  alte Werte:"
      + " a = " + a + ", b = " + b);
    hilf = a;
    a = b;
    b = hilf;
    System.out.println("tausche:  neue Werte:"
      + " a = " + a + ", b = " + b);
  }

  public static void main(String[] argv) {
    long x = 10, y = 12;
    System.out.println("main:     alte Werte:"
      + " x = " + x + ", y = " + y);
    tausche(x, y);
    System.out.println("main:     neue Werte:"
      + " x = " + x + ", y = " + y);
  }
}
```

Parameterübergabe-Mechanismen

Beispiel: Methode `tausche()` - falscher Ansatz

Ausgabe des Programms:

```
main:     alte Werte: x = 10, y = 12
tausche:  alte Werte: a = 10, b = 12
tausche:  neue Werte: a = 12, b = 10
main:     neue Werte: x = 10, y = 12
```

In der Methode `tausche()` werden die beiden Werte, die an die formalen Parameter a und b übergeben werden, richtig vertauscht, wie die mittleren beiden Ausgaben zeigen.

Da die Methode lediglich die *Werte* und nicht die Referenzen der aktuellen Parameter kennt, werden nur die Werte der beiden formalen Parameter vertauscht, die Werte der aktuellen Parameter x und y bleiben unverändert.

Um die Werte der aktuellen Parameter tatsächlich zu tauschen, müssen wir an die *Referenzen* dieser Parameter herankommen.

Objekte sind immer von einem Referenztyp. Im folgenden Beispiel kapseln wir `long`-Werte in einer Klasse Lang.

Parameterübergabe-Mechanismen

Beispiel: Tauschen über Referenzparameter

```
// Hilfsklasse für "call by reference";
// Basisdatentyp long wird in Klasse Lang gekapselt.
class Lang {
   long x;
   // Default-Konstruktor
   Lang() {}
   // "Umwandlung" von long nach Lang
   Lang(long z) { x = z; }
}

class ReferenzParameter {
   static void tausche(Lang a, Lang b) {
      long hilf = a.x;
      a.x = b.x;
      b.x = hilf;
   }

   public static void main(String[] argv) {
      long x = 10, y = 12;
      System.out.println(
         "alte Werte: x = " + x + ", y = " + y);
      Lang xx = new Lang(x);
      Lang yy = new Lang(y);
      tausche(xx, yy);
      x = xx.x;  y = yy.x;
      System.out.println(
         "neue Werte: x = " + x + ", y = " + y);
   }
}
```

Parameterübergabe-Mechanismen

Beispiel: Tauschen über Referenzparameter

Ausgabe des Programms:

```
alte Werte : x = 10, y = 12
neue Werte : x = 12, y = 10
```

Auch hier werden an die Methode wieder nur die Werte der aktuellen Parameter übergeben: dies sind hier zwei Referenzvariablen, die als Wert die *Referenzen* auf die aktuellen Parameter vom Typ Lang enthalten.

Die aktuellen Parameter selbst können wir nicht verändern. Aber über die formalen Parameter können wir die Attribute der übergebenen Objekte ändern.

Diese Form der Parameterübergabe heißt **Referenzübergabe** (engl. *call by reference*).

Parameterübergabe-Mechanismen

Wenn wir die Koordinaten zweier Punkte tauschen wollen, ist keine Hilfsklasse erforderlich, da `Punkt` schon eine Klasse - und damit ein Referenztyp - ist.

Beispiel: zwei Punkte tauschen

```
class TauschePunkte {
   public static void main(String[] argv) {
      Punkt p1 = new Punkt(1, 2, "P1"),
            p2 = new Punkt(3, 4, "P2");
      p1.ausgeben(); System.out.println();
      p2.ausgeben(); System.out.println();
      tausche(p1,p2);
      System.out.println("getauschte Punkte");
      p1.ausgeben(); System.out.println();
      p2.ausgeben(); System.out.println();
   }

   // Zwei Punkte tauschen.
   // Nur die Koordinaten werden getauscht.
   static void tausche(Punkt p1, Punkt p2) {
      Punkt hilf = new Punkt(0, 0, "hilf");
      hilf.x = p1.x; hilf.y = p1.y;
      p1.x = p2.x; p1.y = p2.y;
      p2.x = hilf.x; p2.y = hilf.y;
   }
}
```

Parameterübergabe-Mechanismen

Eine Methode kann *einen* Ergebniswert über den Rückgabewert an den Aufrufer zurückgeben.
Wenn sie mehr als einen Ergebniswert liefern soll, brauchen wir wieder die Referenzübergabe.

Beispiel: Ergebnisparameter

```
class Lang
{
   long x;
}

class ErgebnisParameter
{
   static long sum_max
      (long a, long b, Lang max)
   {
      max.x = (a>b ? a : b);
      return a + b;
   }

   public static void main(String[] argv)
   {
      long x = 10, y = 12, summe;
      Lang maximum = new Lang();
      summe = sum_max(x, y, maximum);
      System.out.println("Summe = " + summe +
         ", Maximum = " + maximum.x);
   }
}
```

Das Schlüsselwort `this`

Die Berechnung des Maximums zweier Punkte sollte eine Methode der Klasse Punkt sein. Wenn wir diese Methode implementieren wollen, tritt ein Problem auf:

```
class Punkt {
  int x, y;
  String name;
  // ...
  Punkt maximum(Punkt p) {
    if(laenge() > p.laenge())
      return  ?????
    else
      return p;
  }
}
```

Die Methode wollen wir wie folgt aufrufen können:

```
Punkt m = p1.maximum(p2);
```

An der mit Fragezeichen markierten Stelle wollen wir das Objekt zurückgeben, auf das diese Methode angewendet wird. Dessen Name ist aber nicht bekannt.

Das Schlüsselwort **`this`** liefert eine Referenz auf das aktuelle Objekt, auf das die Methode angewendet wird.

Das Schlüsselwort `this`

Beispiel: Rückgabewert `this`

```
class Punkt {
  int x, y;
  String name;
  // ...
  Punkt maximum(Punkt p) {
    if(laenge() > p.laenge())
      return this;
    else
      return p;
  }
}
```

Klassen und Objekte

Das Schlüsselwort `this`

Die Parameternamen der Konstruktoren der Klasse `Punkt` sagen bisher nichts über ihre Verwendung aus.

```
class Punkt {
  int x, y;
  String name;
  // ...
  Punkt(int a, int b, String n) {
    x = a; y = b; name = n;
  }
}
```

Es wäre klarer, wenn bei den Konstruktoren die Parameternamen auch `x`, `y` und `name` lauten würden - genau wie die Objektattribute.

```
class Punkt {
  int x, y;
  String name;
  // ...
  Punkt(int x, int y, String name) {
    x = x; y = y; name = name;
  }
}
```

Auf diese Weise können wir leider nicht auf die zu initialisierenden Attribute zugreifen. Sie sind im Konstruktor von den Parameternamen *verdeckt*.

Das Schlüsselwort `this`

Das Schlüsselwort `this` ermöglicht auch den Zugriff auf verdeckte Objektattribute.

Beispiel: Zugriff auf verdeckte Attribute mit `this`

```
class Punkt {
  int x, y;
  String name;
  // ...
  Punkt(int x, int y, String name) {
    this.x = x; this.y = y; this.name = name;
  }
}
```

Das Schlüsselwort `this`

Beim Überladen des Konstruktors der Klasse Punkt wurde bisher jeweils der gesamte Code des Konstruktorrumpfs wiederholt.

Einfacher - insbesondere bei komplexeren Konstruktoren - wäre es, wenn wir einen anderen Konstruktor direkt aufrufen könnten.

In Java kann man den "Methodennamen" **`this`** für den expliziten Aufruf von Konstruktoren innerhalb anderer Konstruktoren verwenden.

Ein **expliziter Konstruktoraufruf** hat die folgende Syntax:

```
this ( Ausdrucks-Liste ) ;
```

oder

```
super ( Ausdrucks-Liste ) ;
```

Die **Ausdrucks-Liste** enthält einen oder mehrere **Ausdrücke**, die durch Komma getrennt sind.

Das Schlüsselwort `this`

Beispiel: Konstruktor `this(...)`

```
class Punkt
{
  int x, y;
  String name;

  Punkt(int x, int y, String name)
  { this.x = x; this.y = y; this.name = name; }

  Punkt()
  { this(0, 0, "unbenannt"); }

  Punkt(String name)
  { this(0, 0, name); }

  Punkt(int x, int y)
  { this(x, y, "unbenannt"); }

  // ...
}
```

Der Aufruf von `this(...)` darf nur in Konstruktoren enthalten sein; er muss dann die *erste* Anweisung im Konstruktorrumpf sein.

Klassen- und Objektattribute

Jedes Objekt hat für alle in der Klasse definierten Attribute eigene Kopien. Den Programmcode für die Methoden gibt es dagegen pro Klasse nur einmal.

Manchmal ist es notwendig, dass alle Objekte einer Klasse ein bestimmtes Attribut gemeinsam benutzen. Die Klasse `Punkt` könnte z. B. ein gemeinsames Attribut haben, das die Anzahl aller aktuell vorhandenen Punkte zählt.

Ein solches gemeinsames Attribut, das pro Klasse genau einen Speicherplatz belegt, wird mit dem Schlüsselwort **`static`** deklariert.

Ein statisches Attribut heißt **Klassenattribut** - es existiert pro Klasse genau einmal.

Ein nicht-statisches Attribut heißt **Objektattribut** - jedes Objekt hat seine eigene Kopie dieses Attributs.

Klassen- und Objektattribute

Beispiel: Klassenattribut `AnzahlPunkte`

```
class Punkt {
   int x, y;
   String name;
   static int AnzahlPunkte = 0;

   Punkt(int x, int y, String name) {
      this.x = x; this.y = y; this.name = name;
      AnzahlPunkte++;
   }

   Punkt()
   {  this(0, 0, "unbenannt");  }

   Punkt(String n)
   {  this(0, 0, n);  }

   Punkt(int a, int b)
   {  this(a, b, "unbenannt");  }

   // weitere Methoden wie gehabt
   // ...

   void zeige() {
      System.out.println(name + "(" + x + "," + y +
         "), Anzahl= " + AnzahlPunkte);
   }

   void zeige(String txt) {
      System.out.print(txt + ": ");
      zeige();
   }
}
```

Klassen- und Objektattribute

Beispiel: Klassenattribut `AnzahlPunkte`

```
class PunktTest
{
   public static void main(String[] argv)
   {
      // Zugriff über den Klassennamen:
      System.out.println("AnzahlPunkte = " +
         Punkt.AnzahlPunkte);

      Punkt p1 = new Punkt(1, 2, "p1"),
            p2 = new Punkt(3, 3, "p2");
      p1.zeige();
      p2.zeige();

      Punkt max,
            p3 = new Punkt();
      p3.zeige();
      max = p1.maximum(p2).maximum(p3);
      max.zeige("Maximaler Punkt");

      // Zugriff über Objektnamen (schlechter Stil):
      System.out.println("Es gibt jetzt " +
         p1.AnzahlPunkte + " verschiedene Punkte");
      System.out.println("Es gibt jetzt " +
         p2.AnzahlPunkte + " verschiedene Punkte");
   }
}
```

Klassenmethoden

Auch Methoden können mit dem Modifizierer `static` gekennzeichnet werden.

Statische Methoden heißen **Klassenmethoden**;
nicht-statische Methoden heißen **Objektmethoden**.

Eine Klassenmethode kann nur auf Klassenattribute und Klassenmethoden der Klasse zurückgreifen.

Da Klassenmethoden sich immer auf eine *Klasse* und nicht auf ein Objekt beziehen, darf in ihrem Rumpf das Schlüsselwort `this` nicht verwendet werden. Es gibt keine `this`-Referenz, denn es gibt ja kein spezifisches Objekt, auf dem gearbeitet werden kann.

Klassenmethoden

Beispiel: Klassenmethode `maximum()`

```
class Punkt {
   int x, y;
   String name;
   static int AnzahlPunkte = 0;

   Punkt(int x, int y, String name) {
      this.x = x; this.y = y; this.name = name;
      AnzahlPunkte++;
   }

   // weitere Konstruktoren und Methoden
   // ...

   // Objektmethode maximum(), wie gehabt
   Punkt maximum(Punkt p) {
      if (laenge() > p.laenge())
         return this;
      else
         return p;
   }

   // Klassenmethode maximum()
   static Punkt maximum(Punkt p1, Punkt p2) {
      if (p1.laenge() > p2.laenge())
         return p1;
      else
         return p2;
   }
}
```

Klassenmethoden

Beispiel: Klassenmethode `maximum()`

```
class PunktTest
{
   public static void main(String[] argv)
   {
      Punkt p1 = new Punkt(1, 2, "p1"),
            p2 = new Punkt(3, 3, "p2"),
            max;

      // Objektmethode
      max = p1.maximum(p2);
      max.zeige("Maximaler Punkt (Objektmethode)");

      // Klassenmethode
      max = Punkt.maximum(p1, p2);
      max.zeige("Maximaler Punkt (Klassenmethode)");
      max = p1.maximum(p1, p2);
      max.zeige("Maximaler Punkt (Klassenmethode)");
   }
}
```

Klassenbezogene Initialisierungsblöcke

Die Initialisierung von statischen Attributen geschieht bei der Vereinbarung

```
static int AnzahlPunkte = 0;
```

und / oder in einem separaten Block, dem **klassenbezogenen** (statischen) **Initialisierungsblock**

```
static int AnzahlPunkte;
// ...
static
{
   AnzahlPunkte = 0;
}
```

Eine Klasse kann mehrere statische Initialisierungsblöcke haben; diese werden dann nacheinander in der aufgeschriebenen Reihenfolge ausgeführt.

Klassenbezogene Initialisierungsblöcke

Beispiel: statischer Initialisierungsblock

```
class Punkt {
  int x, y;
  String name;
  static int AnzahlPunkte, MaxPunkte;

  static {
    AnzahlPunkte = 0;
    MaxPunkte = 4;
  }

  // Hauptkonstruktor. Er überprüft auf MaxPunkte.
  Punkt(int x, int y, String name) {
    if (AnzahlPunkte < MaxPunkte)
      AnzahlPunkte++;
    else {
      System.out.println(
        "Zu viele Punkte. Fehler bei Punkt " + name);
      return;
    }
    this.x = x;
    this.y = y;
    this.name = name;
  }

   // Rest wie gehabt
}
```

Finalisierung eines Objekts

Java führt selbständig eine Speicherbereinigung durch; es ist daher nicht notwendig, Objekte explizit freizugeben.

Objekte werden mit `new` erzeugt; es gibt jedoch kein `delete` (wie in C++), sondern Objekte werden automatisch gelöscht, wenn es keine Referenz mehr darauf gibt.

Nach dem Verlassen des Definitionsbereichs eines Objekts, z. B. der Methode in der das Objekt definiert ist, gibt es keine Referenz auf das Objekt mehr.

Eine Referenz kann auch explizit gelöscht werden:

```
Punkt p1 = new Punkt(1, 2, "p1");
// Arbeiten mit p1.
...
// Von jetzt an wird p1 nicht mehr
// benutzt, aber es folgt lange keine
// Rückkehr aus der Methode.
p1 = null;
```

Finalisierung eines Objekts

Neben der Speicherplatzfreigabe sind bei der Finalisierung eines Objekts manchmal noch weitere Aufgaben zu erledigen, z. B. die Freigabe von Netzwerkverbindungen und das Schließen von Dateien.

In der Klasse `Punkt` sollte beim Freigeben eines `Punkt`-Objekts auch der Zähler `AnzahlPunkte` erniedrigt werden.

Solche Aufgaben können wir in der speziellen Methode

```
public void finalize()
```

festlegen. Diese Methode wird automatisch vor dem Aufruf des garbage collector aufgerufen.

Die Reihenfolge der Freigabe von Objekten durch den garbage collector ist nicht definiert. Die Methoden `finalize()` verschiedener Objekte werden daher in einer zufälligen Reihenfolge aufgerufen.

Die Methode `finalize()` kann auch explizit aufgerufen werden:

```
objekt.finalize();
```

Finalisierung eines Objekts

Beispiel: `finalize()` für die Klasse `Punkt`

```
class Punkt {

  // Finalisierer mit Testausdrucken,
  // die anzeigen, wann er aufgerufen wird.

  public void finalize()
  {
    System.out.println(
      "\t>>>Aufruf     finalize(): " + AnzahlPunkte);
    AnzahlPunkte--;
    System.out.println(
      "---. neue AnzahlPunkte : " + AnzahlPunkte);
  }

  // Rest wie gehabt

}
```

Finalisierung eines Objekts

Beispiel: `finalize()` für die Klasse `Punkt`

```
class PunktTest
{
   public static void main(String[] argv)
   {
      Punkt p1 = new Punkt(1, 2, "p1"),
            p2 = new Punkt(3, 3, "p2"),
            max;

      // Objektmethode
      max = p1.maximum(p2);
      max.zeige("Maximaler Punkt (Objektmethode)");

      // Klassenmethode
      max = Punkt.maximum(p1, p2);
      max.zeige("Maximaler Punkt (Klassenmethode)");

      p1.finalize();
      p2.finalize();

      p1 = new Punkt();
      System.out.println("AnzahlPunkte = " +
         Punkt.AnzahlPunkte);
   }
}
```

Finalisierung eines Objekts

Beispiel: `finalize()` für die Klasse `Punkt`

Ausgabe des Programms:

```
Maximaler Punkt (Objektmethode): p2(3,3), Anzahl= 2
Maximaler Punkt (Klassenmethode): p2(3,3), Anzahl= 2
        >>>Aufruf finalize(): 2
---. neue AnzahlPunkte : 1
        >>>Aufruf finalize(): 1
---. neue AnzahlPunkte : 0
AnzahlPunkte = 1
```

Wenn die beiden Zeilen

```
p1.finalize();
p2.finalize();
```

gelöscht werden, dann wird folgende Ausgabe erzeugt:

```
Maximaler Punkt (Objektmethode): p2(3,3), Anzahl= 2
Maximaler Punkt (Klassenmethode): p2(3,3), Anzahl= 2
AnzahlPunkte = 3
```

Der garbage collector wurde offensichtlich nicht aufgerufen.

Die Methode `toString()`

Wenn eine Klasse eine Methode

```
public String toString()
```

bereitstellt, so wird diese Methode immer dann aufgerufen, wenn ein Objekt dieser Klasse an einer Stelle verwendet wird, wo ein `String`-Objekt für einen `+` oder `+=` Operator erwartet wird.

Die Methode `toString()` unterstützt Zeichenkettenoperationen wie die direkte Ausgabe mit `System.out.println()` oder die Aneinanderreihung mit den Operatoren `+` und `+=`.

Die Methode `toString()`

Beispiel: `toString()` für die Klasse `Punkt`

```
class Punkt
{
   // alles wie gehabt

   void zeige()
   {
      System.out.println(
         name + "(" + x + "," + y + ")"
         + ", Anzahl= " + AnzahlPunkte);
   }

   // Der Rückgabewert der Methode toString()
   // ist eine Zeichenkette, die alle Informationen
   // über das jeweilige Punkt-Objekt enthält.

   public String toString()
   {
      return
         name + "(" + x + "," + y + ")"
         + ", Anzahl= " + AnzahlPunkte;
   }
}
```

Die Methode `toString()`

Beispiel: `toString()` für die Klasse `Punkt`

```
class PunktTest
{
   public static void main(String[] argv)
   {
      Punkt p1 = new Punkt(1, 2, "p1"),
            p2 = new Punkt(3, 3, "p2");

      p1.zeige();

      System.out.println(p1.toString());
      System.out.println(p1);

      System.out.println(
         "Hier kommt Punkt p2: " + p2);

      String s = "tralala: ";
      s += p1;
      System.out.println(s);
   }
}
```

Ausgabe des Programms:

```
p1(1,2), Anzahl= 2
p1(1,2), Anzahl= 2
p1(1,2), Anzahl= 2
Hier kommt Punkt p2: p2(3,3), Anzahl= 2
tralala: p1(1,2), Anzahl= 2
```