# 2.9 Klassen und Objekte

Mit einer **Klassendeklaration** definiert man neue Referenztypen und legt gleichzeitig deren Implementation fest.

Jede Klasse (außer **`Object`**) ist implizit Subklasse der Klasse **`Object`**; es gibt also eine gemeinsame Wurzel der Java-Vererbungshierarchie.

# Was zu einer Klasse gehört

Im Rumpf einer Klasse kann man folgendes deklarieren:

- **Variablen** eines beliebigen Typs, in denen man Objektzustände speichert,
- **Methoden**. Das sind die Operationen, die auf Objekte der Klasse angewendet werden können, die also das Objektverhalten implementieren,
- **Konstruktoren**, das sind spezielle Methoden, mit denen man die Variablen initialisiert,
- **`Static`-Initialisierer**. Das sind spezielle Anweisungsfolgen, die nach dem Laden der Klasse (ebenfalls zum Zweck der Initialisierung) einmal ausgeführt werden, und
- **eingebettete Klassen**, das sind Klassen, die einen neuen Referenztyp deklarieren; meistens handelt es sich dabei um Hilfsklassen, die man nur lokal benötigt.

# Klassenvariablen und Instanzvariablen

Variablen können als **Klassenvariablen**, die Java einmal pro Klasse anlegt, oder als **Instanzvariablen**, die für jedes Objekt neu erzeugt werden, deklariert werden.

Ebenso ist es möglich, eine Methode als **Klassenmethode**, die ohne Objekt existiert, oder als **Instanzmethode**, die immer für ein bestimmtes Objekt aufgerufen wird, zu deklarieren. Der Unterschied besteht jeweils in der Verwendung des Modifizierers **`static`**.

# Beispiel für die Deklaration einer Klasse

```
class Ticker {
  int min, sek, tsdSek;
  void stelle(int i, int j, int k) {
    min = i;
    sek = j;
    tsdSek = k;
  }
  void tick() {
    if (++tsdSek == 1000) {
        tsdSek = 0;
        if (++sek == 60){
            sek = 0;
            min++;
        }
    }
  }
  void zeigeAn() {….formatierte Ausgabe}
}
```

# Klassenreferenz und Klassenerzeugung

Nach der Deklaration der **`Ticker`**-Klasse ist es möglich, ihren Namen zur Deklaration von Variablen wie jeden anderen Typnamen, z.B. **`int`**, **`double`** usw., zu benutzen. Es muss aber beachtet werden, dass es sich um einen Referenztyp handelt, dass also mittels

```
Ticker t;
```

kein `Ticker`-Objekt erzeugt wird, sondern dass t lediglich eine Variable ist, die eine Referenz auf ein solches Objekt enthalten kann. Ein Objekt wird erst durch ein explizites **`new`** in einem „Instanzerzeugungs-Ausdruck“, z. B.

```
t = new Ticker();
```

erzeugt.

Nach der Objekterzeugung sind die Instanz- und Klassenvariablen mit Standardwerten initialisiert. Im Beispiel haben alle drei Variablen **`min`**, **`sek`** und **`tsdSek`** den Wert 0.

# Zugriff auf Klassenelemente aus der eigenen Klasse

Innerhalb der Klassendeklaration kann man auf die Klassenelemente mit ihrem Bezeichner zugreifen. Im `Ticker`-Beispiel wird in allen drei Methoden auf alle Instanzvariablen zugegriffen. Auch das Aufrufen einer Methode innerhalb einer anderen Methode ist möglich:

```
class X {
  void g() {
    …
    f();
  }
  void f() {…}
}
```

# Zugriff auf Klassenelemente aus fremden Klassen

Code in anderen Klassen greift auf Instanzvariablen und Instanzmethoden mit einer Objektreferenz unter Verwendung von "." zu (als Interpunktionszeichen, das in diesem Kontext auch als Zugriffsoperator bezeichnet wird).

**Beispiel:**

```
Ticker t1, t2;
t1 = new Ticker();
t2 = new Ticker();
t1.min = 10;
t1.sek = 23;
t1.tsdSek = 5;
t2.stelle(10,23,5);
for (int i=0; i<1500; i++)
     t2.tick();
```

Hier werden zwei Ticker-Objekte erzeugt, die beide ihre eigenen Instanzvariablen haben.

# Die Zugriffsrechte `public`, `protected`, `private`

Ob man auf die Klassenelemente von außerhalb ihrer Klasse zugreifen kann, hängt davon ab, welche Zugriffsrechte vergeben wurden und von wo aus der Zugriff erfolgen soll. Zur expliziten Festlegung von Zugriffsrechten verwendet man die Modifizierer **`public`**, **`protected`** oder **`private`**. Üblicherweise spezifiziert man Instanzvariablen als **`private`**, um die in ihnen gespeicherten Werte oder die von ihnen referenzierten Objekte zu „kapseln“, also:

```
class Ticker {
  private int min, sek, tsdSek;
  …
}
```

Auf die Variablen kann dann nur noch von Code innerhalb der **`Ticker`**-Deklaration zugegriffen werden, und im Beispiel müssen die drei Zuweisungen durch einen Aufruf **`t1.stelle(10,23,5);`** ersetzt werden.

# Das Schlüsselwort `this` (1)

Der Name einer Instanzvariablen kann durch eine lokale Variable oder einen Methoden- oder Konstruktorparameter desselben Namens **verdeckt** werden.

Unter Verwendung des Schlüsselworts **`this`** kann man dennoch auf die verdeckte Variable zugreifen. Dies dient nicht der Verständlichkeit des Codes, ist aber bei Konstruktoren gängige Java-Praxis.

# Das Schlüsselwort `this` (2)

Zum Beispiel wird im Folgenden der Wert des Parameter **`i`** bzw. der lokalen Variablen **`i`** an die Instanzvariable **`i`** zugewiesen:

```
class X {
  private int i;
  X(int i) {
    this.i = i;
  }
  void f() {
    int i = 10;
    this.i += i;
  }
}
```

# Klassenvariablen und Instanzvariablen (1)

Wenn man eine Variable als **`static`** deklariert, wird sie zur **Klassenvariablen**. Dies bedeutet, dass es genau eine derartige Variable gibt, gleichgültig, wieviele Objekte der Klasse existieren. Klassenvariablen werden beim Laden der Klasse erzeugt.

Instanzvariablen sind Variablen, die nicht als **`static`** deklariert wurden. Sie werden für jedes Objekt bei seiner Erzeugung angelegt.

Wird die Klasse **`Ticker`** beispielsweise um eine Klassenvariable **`anzTicks`** erweitert, die immer die Anzahl der **`tick`**-Aufrufe enthalten soll, so sieht man, dass innerhalb der Klassendeklaration auf eine Klassenvariable wie auf eine Instanzvariable einfach mit ihrem Bezeichner zugegriffen werden kann:

# Klassenvariablen und Instanzvariablen (2)

```
class StatTicker {
  static long anzTicks = 0;
  private int min, sek, tsdSek;
  void stelle(int i, int j, int k) {… unverändert}
  void tick() {
    anzTicks++;
    … Rest unverändert
    }
    void zeigeAn() {… unverändert}
}
```

# Zugriff auf Klassenvariablen aus fremden Klassen (1)

An dem folgenden Anwendungsbeispiel erkennt man weiterhin, dass von außerhalb der Klasse mittels einer Objektreferenz oder sinnvoller, weil eine Klassenvariable unabhängig von Objekten existiert, auch mit dem Klassennamen anstelle einer Referenz zugegriffen werden kann.

Der Zugriff auf eine Klassenvariable über eine Objektreferenz ist zwar möglich, aber nicht zu empfehlen, da hier der Eindruck eines Objektbezugs erweckt wird, der bei Klassenvariablen nicht existiert.

# Zugriff auf Klassenvariablen aus fremden Klassen (2)

```
System.out.println("Anzahl ticks: "
  + StatTicker.anzTicks);  // Zugriff ohne Objekt

StatTicker s1 = new StatTicker(), s2 = new StatTicker();

for (int i=0; i<21500; i++)

  s1.tick();

for (int i=0; i<821500; i++)

  s2.tick();

System.out.println(" Anzahl ticks: "
  + StatTicker.anzTicks + "\t"
  + s1.anzTicks + "\t" + s2.anzTicks);   // Zugriff mit Objekt
```

# Die Initialisierung von Klassen- und Instanzvariablen

- Die Initialisierung von Klassenvariablen (also von als **`static`** deklarierten Variablen), wird genau einmal, beim Initialisieren der Klasse nach dem Laden, vorgenommen.
- Instanzvariablen (also Variablen, die nicht als **`static`** deklariert sind), werden jedes Mal, wenn ein Objekt der Klasse dynamisch erzeugt wird, initialisiert.
- Der Initialisierer einer Instanzvariablen darf keine nachfolgend deklarierten Instanzvariablen und auch nicht die gerade deklarierte Variable enthalten.

```
class X {
  double x = i;       // Fehler: Vorwärtsreferenz auf Instanzvariable
  int i = 5;
  int j = j+1;        // Fehler: j wird hier gerade deklariert
  double y = z;       // korrekt: Vorwärtsreferenz auf Klassenvariable
  static double z = -1.2345;
  …
}
```

# Symbolische Konstanten in Klassen

Instanz- und Klassenvariablen können mit dem bereits eingeführten **`final`**-Modifizierer als symbolische Konstanten deklariert werden. In diesem Fall sollte die Deklaration einen **Initialisierer** enthalten.

Alternativ kann für **`final`**-Instanzvariablen in jedem Konstruktor bzw. für **`final`**-Klassenvariablen in einem **`static`**-Initialisierer **genau eine Zuweisung** vorgenommen werden.

Der Wert derartiger „Variablen“ kann nach ihrer Initialisierung bzw. der ersten Zuweisung nicht mehr verändert werden.

# Beispiele für symbolische Konstanten in Klassen

```
class Rechnung {
  static long nr = 0;
  final long nummer = ++nr;
  final Date datum = new Date();
  …
}
```

Die Nummer der Rechnung und das Rechnungsdatum sind als konstante Instanzvariablen deklariert, die bei der Erzeugung eines **Rechnung**-Objekts mit der nächsten Nummer bzw. dem aktuellen Datum initialisiert werden.

# Deklaration von Methoden

Eine Methodendeklaration besteht aus bis zu sechs Teilen:

- **optionalen Modifizierern**, wie `public`, `abstract`, `final` usw., die spezielle Attribute der Methode festlegen,
- dem **Typ des Resultats**, das bei einem Methodenaufruf berechnet wird,
- einem **Bezeichner**, mit dem die Methode benannt wird,
- der Liste der **Methodenparameter**, mit denen die Werte, die bei einem Aufruf zu übergeben sind, spezifiziert werden,
- einer **optionalen** `throws`**-Klausel**, die anzeigt, welche Ausnahmen durch einen Aufruf der Methode ausgeworfen werden können, und
- dem **Methodenrumpf**, der – in { und } eingeschlossen – die Folge der Anweisungen enthält, die die Funktionalität der Methode beinhalten.

# Aufruf von Methoden

Ein **Methodenaufruf** ist ein elementarer Ausdruck. Bei jedem Aufruf einer Methode werden die Parameter neu erzeugt und mit den Werten der aktuellen Argumente initialisiert, bevor mit der Ausführung des Methodenrumpfs begonnen wird.

Die Parameter erhalten als Geltungsbereich den gesamten Methodenrumpf und werden ansonsten wie lokale Variablen behandelt; sie dürfen nicht durch Redeklarationen verdeckt werden. Die Argumente werden als Werte übergeben und können daher durch den Aufruf nicht modifiziert werden. Java kennt also nur einen „call by value“!

# Veränderung von Parametern durch Methoden (1)

Ob sich die Veränderung eines Parameters innerhalb des Methodenrumpfs auf die aufrufende Methode auswirken kann, hängt davon ab, ob es sich bei seinem Typ um einen elementaren Typ oder einen Referenztyp handelt:

```
class Parameter {
  static void m(int i, Zaehler z) {
    i++;
    z.inkrementiere();
  }
  public static void main(String[] args) {
    int a = 2;
    Zaehler b = new Zaehler();
    b.wert(2);
    System.out.println(a + " " + b.wert());
    m(a,b);
    System.out.println(a + " " + b.wert());
  }
}
```

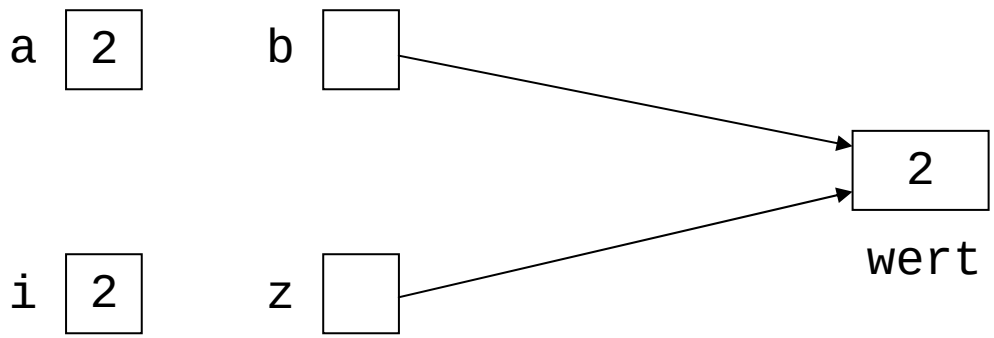
# Veränderung von Parametern durch Methoden (2)

Beim Aufruf von **`m`** werden die Parameter **`i`** und **`z`** erzeugt, in **`i`** wird der Wert von **`a`**, also 2, in **`z`** wird die Referenz auf das `Zaehler`-Objekt **`b`** (die Adresse dieses Objekts) kopiert.

Das Inkrementieren von **`i`** ändert lediglich den Parameterwert. Dagegen verweisen der Parameter **`z`** und die Instanzvariable **`b`** auf dasselbe Objekt. Dieses wird inkrementiert. Nach dem Aufruf wird also **`2 3`** ausgegeben. Das heißt: Obwohl auch Referenzen als Wert übergeben werden und sich am Referenzwert nichts ändert, wird hier dennoch der Effekt eines „call by reference“ erzielt.

Die folgende Abbildung zeigt die Variablen und Parameter nach der Parameterinitialisierung und zu Beginn der Ausführung des Methodenrumpfs.

# Veränderung von Parametern durch Methoden (3)




Nach der Ausführung des Methodenaufrufs werden die Parameter **`i`** und **`z`** vom Garbage-Collector zerstört.

# Die `return`-Anweisung (1)

Eine **`return`**-Anweisung beendet die Ausführung einer Methode bzw. eines Konstruktors und verzweigt den Kontrollfluss zu der aufrufenden Methode bzw. zum aufrufenden Konstruktor. Sie hat die Form

`return` Ausdruck$_{opt}$ ;

In der Form **`return;`** darf sie nur in Methoden, die mit **`void`** als Ergebnis-"Typ" deklariert wurden, oder in Konstruktoren vorkommen. Der Methodenaufruf liefert dann keinen Wert. Falls eine Methode mit Ergebnistyp **`void`** keine **`return`**-Anweisung enthält, fügt der Java-Compiler implizit als letzte Anweisung ein **`return;`** ein.

# Die `return`-Anweisung (2)

Eine **`return`**-Anweisung mit Ausdruck darf nur in einer Methode stehen, die nicht mit **`void`** als Typ des Ergebnisses deklariert ist. Und umgekehrt muss eine Methode mit einem Ergebnistyp ungleich **`void`** mit einem expliziten **`return`** und der Rückgabe eines Werts beendet werden. Der Aufrufer setzt seine Aktivitäten dann mit der weiteren Auswertung des umgebenden Ausdrucks oder mit der nächsten Anweisung fort.

Bei einem **`return`** mit Ausdruck muss der Typ des Ausdrucks zuweisungskompatibel zum Ergebnistyp der Methode sein, und der Wert des Methodenaufrufs ist dann der Wert dieses Ausdrucks.

# Instanz- und Klassenmethoden

So wie es objekt- und klassenspezifische Variablen gibt, kann man auch den Objektbezug einer Methode dadurch auflösen, dass man sie als **`static`** deklariert. Sie wird dadurch zur **Klassenmethode**.

Dies bedeutet, dass die Methode nicht für ein bestimmtes Objekt, sondern unabhängig von Objekten aufgerufen wird. Wie bei Klassenvariablen benutzt man dann den Klassennamen anstelle eines Instanznamens.

Instanzmethoden sind Methoden, die nicht als **`static`** deklariert wurden.

# Einschränkungen für Klassenmethoden (1)

Klassenmethoden können nur auf Klassenvariablen ihrer Klasse zugreifen.
Sie dürfen das Schlüsselwort **`this`** nicht verwenden.

```
class X {
  int x;
  static int y;
  static void f() {x++;}    // Fehler: x ist nicht static
  static void g() {y++;}    // korrekt: y ist static
  …
}
```

# Einschränkungen für Klassenmethoden (2)

Der Grund für diese Einschränkungen ist offensichtlich: Wenn `f` nicht für ein bestimmtes Objekt aufgerufen wird, ist **`this`** undefiniert: es ist nicht klar, in welchem Objekt die Instanzvariable inkrementiert werden soll.

Klassenmethoden können darüber hinaus nicht als **`abstract`** deklariert werden. Hierauf werden wir in Kapitel 10 noch näher eingehen.

# Das Überladen von Methoden (1)

Mehrere Methoden einer Klasse können denselben Namen erhalten, zum Beispiel:

```
class X {
  void x(int i) {System.out.println(i);}
  void x(double d) {System.out.println(d);}
  …
}
```

Voraussetzung ist dabei lediglich, dass die **Signatur** der Methoden verschieden ist. Die Signatur einer Methode besteht aus dem Namen der Methode und der Anzahl sowie der (geordneten) Folge der Typen der formalen Parameter.

# Das Überladen von Methoden (2)

Eine Klassendeklaration darf nicht mehr als eine Methode mit einer bestimmten Signatur enthalten. Es ist dabei nicht relevant, ob es sich um Klassen- oder Instanzmethoden handelt. Die folgende Klassendeklaration ist daher fehlerhaft:

```
class Punkt {
  int x, y;
  int verschiebe(int dx, int dy)
    {x += dx; y += dy; return dx*dx + dy*dy;}
  void verschiebe(int dx, int dy)        // Fehler
    {x += dx; y += dy;}
}
```

# Ermittlung der aufzurufenden Methode (1)

Welche von mehreren in Frage kommender Methoden beim Aufruf überladener Methoden ausgeführt wird, entscheidet der Compiler bereits beim Übersetzen anhand der aktuellen Argumente. Java untersucht alle Methoden (auch geerbte),

- die den überladenen Methodennamen tragen,
- deren Parameterzahl mit der Argumentanzahl übereinstimmt,
- für die die Argumenttypen des Aufrufs durch elementare Typvergrößerungen oder Vergrößerungen von Referenztypen (also durch die Methodenaufruf-Konversionen) in die Parametertypen konvertierbar sind oder mit ihnen übereinstimmen, und
- die zugreifbar sind. Die von den Modifizierern **`public`**, **`protected`** usw. abhängige Zugreifbarkeit besprechen wir später.

# Ermittlung der aufzurufenden Methode (2)

Unter diesen **aufrufbaren** Methoden wird dann die **spezifischste (**„am besten passende“) Methode dadurch ermittelt, dass Java alle Methodenpaare **`m(S1, …, Sn)`** und **`m(T1,…, Tn)`** miteinander vergleicht und **`m(T1,…, Tn)`** aus der Liste der aufrufbaren Methoden streicht, wenn für alle **`j`** von **`1`** bis **`n`** eine Methodenaufruf-Konversion von **`Sj`** nach **`Tj`** existiert oder **`Sj`** gleich **`Tj`** ist.

Sofern dieser Prozess die aufrufbaren Methoden nicht auf genau eine – die spezifischste – Methode reduziert, ist der Funktionsaufruf mehrdeutig und wird als Fehler diagnostiziert.

# Ermittlung der aufzurufenden Methode (Beispiel 1)

```
class X {
  static void m(double d)
    {System.out.println(“m(double)“);}
  static void m(int i)
    {System.out.println(“m(int)“);}
  public static void main(String[] args) {
    float x = 3.765f;
    X.m(x);
  }
}
```

Es wird **`m(double)`** ausgeführt, da **`m(int)`** nicht aufrufbar ist. **`m(double)`** ist dagegen aufrufbar, denn es existiert eine elementare Typvergrößerung von **`float`** nach **`double`**.

# Ermittlung der aufzurufenden Methode (Beispiel 2)

```
class Y {
  void m(double d)
    {System.out.println(“m(double)“);}
  void m(float f)
    {System.out.println(“m(float)“);}
  public static void main(String[] args) {
    long l = 3765;
    new Y().m(l);
  }
}
```

Hier sind beide Methoden aufrufbar. **`m(float)`** wird ausgeführt, da **`float`** durch elementare Typvergrößerung nach **`double`** konvertierbar ist, **`m(double)`** also ausscheidet.

# Ermittlung der aufzurufenden Methode (Beispiel 3)

```
class Z {
  void m(double d, int i)
    {System.out.println(“m(double, int)“);}
  void m(int i, double d)
    {System.out.println(“m(int, double)“);}
  public static void main(String[] args) {
    Z z = new Z();
    z.m(5,-5);
  }
}
```

In diesem Beispiel sind beide Methoden aufrufbar. Der Aufruf ist mehrdeutig, und die Klassendeklaration wird nicht übersetzt.

# Die Konstruktion von Objekten

Objekte werden in Java stets dynamisch erzeugt. Die Standard-Vorgehensweise ist die Verwendung des Schlüsselwortes **`new`**.

Es ist auch möglich, die völlig analog arbeitende Methode **`newInstance`** zusammen mit dem entsprechenden Klassenliteral einzusetzen und stattdessen die Form `z = (Z).Z.class.newInstance();` zu benutzen. Diese beiden Vorgehensweisen werden als **explizite** Objekterzeugung bezeichnet.

Darüber hinaus können Objekte **implizit** erzeugt werden, beispielsweise

- **`String`**-Objekte zur Aufnahme der Werte von Zeichenketten oder im Zusammenhang mit der Auswertung des Operators "**`+`**",
- Felder, wenn eine Initialisiererliste angegeben ist,
- beliebige Objekte, wenn wir **`clone`** aufrufen, usw.

# Konstruktoren

Ein **Konstruktor** ist eine spezielle Methode, die den Namen der Klasse trägt. Der Konstruktor wird beim Erzeugen von Objekten der Klasse aufgerufen.

Konstruktoren sind keine Klassenelemente und werden demzufolge nicht vererbt. Sie können weder verdeckt noch überschrieben werden. Konstruktoren können nicht **`static`**, **`abstract`**, **`final`**, **`native`** oder **`synchronized`** sein.

In allen anderen Belangen, z. B. den Möglichkeiten, Zugriffsrechte zu spezifizieren, ihren Namen zu überladen oder Ausnahmen auszuwerfen, unterscheiden sie sich nicht von anderen Methoden.

# Beispiel für Konstruktoren für die Klasse `Ticker` (1)

```
class Ticker {
  private int min, sek, tsdSek;
  Ticker() {
    min = sek = tsdSek = 0;
  }
  Ticker(int min, int sek, int tsdSek) {
    this.min = min;
    this.sek = sek;
    this.tsdSek = tsdSek;
  }
  ... sonst alles unverändert
}
```

# Beispiel für Konstruktoren für die Klasse `Ticker` (2)

Es wird bei einer Anwendung der Art

```
Ticker t1 = new Ticker(),
       t2 = new Ticker(10,23,5);
```

zur Erzeugung des ersten, von **`t1`** referenzierten Objekts der erste Konstruktor **`Ticker()`** aufgerufen, und das zweite Objekt wird mittels **`Ticker(int, int, int)`** konstruiert.

# Der Standard-Konstruktor

Wenn wir eine Klasse `X` ohne Konstruktor deklarieren, erzeugt Java einen **Standard-Konstruktor** der Form:

```
X() {super();}
```

Sofern wir mindestens einen Konstruktor implementieren, entfällt diese implizite Deklaration.

Auch ein von uns deklarierter Konstruktor ohne Parameterliste, wie oben der erste `Ticker`-Konstruktor, wird oft als Standard-Konstruktor bezeichnet.

# Keine Instanzvariablen in Konstruktoren (1)

Für die expliziten Aufrufe von Konstruktoren mittels **`this`** gilt die Einschränkung, dass sie keine Instanzvariablen oder –methoden verwenden dürfen.

```
class X {
  final int i = 10;
  final static int j = 10;
  int f() {return i;}
  X() {this(i);}                    // Fehler: i ist Instanzvariable
  X(double d) {this(f());}          // Fehler: f ist Instanzmethode
  X(String s) {this(j);}
  X(int a) {…}
}
```

# Keine Instanzvariablen in Konstruktoren (2)

Der Grund für diese Einschränkungen ist, dass das Objekt bei diesem Stadium der Initialisierung noch nicht weit genug konstruiert ist, um sicher auf Instanzvariablen oder –methoden zugreifen zu können.

Die Klassenvariable **`j`** dagegen ist zum Zeitpunkt des Aufrufs **`this(j)`** bereits erzeugt und initialisiert.

# `static` Initialisierer

Es ist möglich, innerhalb einer Klasse **`static`**-Initialisierer zu deklarieren. Dies sind Codeblöcke, die auf das Schlüsselworte **`static`** folgen:

`static` Block

Diese Codeblöcke werden **genau einmal** ausgeführt, wenn die Klasse geladen wird. Die Initialisierung von Klassenvariablen und das Auswerten von **`static`**-Initialisierern wird in der Reihenfolge vorgenommen, in der sie in der Klassedeklaration aufeinander folgen.

Mit dem **`static`**-Initialisierer können Klassenvariablen, die Felder, Mengen oder Listen sind, mit Werten versehen werden. Unspezifiziert **`final`** deklarierte Klassenvariablen müssen auf diese Weise initialisiert werden.

# Objektfreigabe (Objektzerstörung)

Die Freigabe des für ein Objekt reservierten Speicherplatzes erfolgt automatisch durch den Java-**Gargabe-Collector**. Hierbei handelt es sich um einen Thread niedriger Priorität, der die existierenden Objekte periodisch daraufhin überprüft, ob sie noch referenziert werden.

Wenn keinerlei Referenzen auf ein Objekt mehr existieren, kann dieses zerstört und sein Speicherplatz anderweitig verwandt werden.

Mit der Methode **`finalize`**, die jede Klasse aus der gemeinsamen Superklasse **`Object`** erbt, steht eine Methode zur Verfügung, die wir für jede Klasse implementieren können und die aufgerufen wird, unmittelbar bevor ein Objekt gelöscht wird.

**`finalize`** muss dazu wie folgt deklariert werden:

```
Protected void finalize() throws Throwable {…}
```

# 2.10 Subklassen, Superklassen, Vererbung

Wie schon erwähnt, erlaubt Java die Deklaration von **Klassenhierarchien**.

Eine Klasse **Y** wird zur **Subklasse** einer anderen Klasse **X**, indem man bei ihrer Deklaration die Superklasse **X** nach dem Schlüsselwort **`extends`** spezifiziert:

```
class Y extends X {…}
```

# Beispiel für eine Subklasse

```
class SignalTicker extends Ticker {
  boolean aktiv;
  int minSig, sekSig, tsdSekSig;
  static final char SIGNAL = '\u0007';
  void sigTick() {
    tick();
    if (aktiv)
        if (min == minSig && sek == sekSig &&
            tsdSek == tsdSekSig) {
            System.out.print(SIGNAL);
            aktiv = false;
        }
    }
}
```

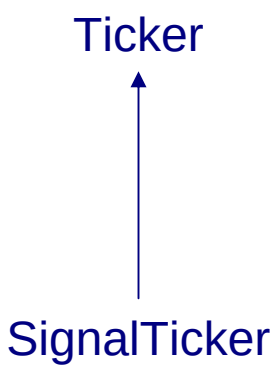
# Anwendungsbeispiel

Eine erste Anwendung könnte beispielsweise so aussehen:

```
SignalTicker  s1 = new SignalTicker(),
  s2 = new SignalTicker();
s2.aktiv = s1.aktiv = true;
s1.tsdSekSig = 10;
s2.tsdSekSig = 750;
for (int i=0; i<1500; i++) {
  s1.sigTick();
  s2.sigTick();
}
```

# Darstellung im Klassendiagramm

Vererbungsbeziehungen stellen wir im Folgenden durch einen Pfeil dar, der in Richtung der Superklasse zeigt:



# Umwandlung von Referenztypen (Upcast)

Wenn **Y** Subklasse von **X** ist, können Referenzen auf **Y** bei Zuweisungen und Methodenaufrufen implizit in Referenzen auf **X vergrößert** werden. Diese Typvergrößerung ist möglich, weil jedes **Y**-Objekt über dieselben (zugreifbaren) Elemente wie ein **X**-Objekt verfügt und somit von seinen Zuständen und von seinem Verhalten her gesehen ein **X**-Objekt ist.

**Beispiel** für die Klasse `UpCasts`:

```
class UpCasts {
  static void f(Ticker t) {
    t.zeigeAn();
  }
public static void main(String[] args) {
    Ticker t = new Ticker();
    SignalTicker s = new SignalTicker();
```

(Fortsetzung)

# Beispiel für die Klasse UpCasts

```
    s.stelle(2,2,571);
    f(t);
    f(s);                    // Typvergrößerung
    t = s;
    t.stelle(1,1,25);
    f(t);
    f(s);                    // Typvergrößerung
  }
}
```

Die Ausgabe ist durch die f ‘s ist wie folgt:

```
00,00,000
02,02,571
01,01,025
01,01,025
```

# Umwandlung von Referenztypen (Downcast)

Der umgekehrte Fall der Verkleinerung von Referenztypen erfordert einen expliziten Cast, den sogenannten "**Down-Cast**". Hier wird zur Laufzeit geprüft, ob die Referenz tatsächlich auf ein Objekt der Subklasse verweist; anderenfalls wird eine Ausnahme des Typs **`ClassCastException`** ausgeworfen. Zum Beispiel:

```
class DownCasts {
  public static void main(String[] args) {
    Ticker t1 = new Ticker(), t2 = new Ticker();
    SignalTicker  s1 = new SignalTicker(),
                  s2 = new SignalTicker();
    s1.stelle(2,2,571);
    t1 = s1;
    s2 = (SignalTicker)t1;  // Downcast auf SignalTicker
    System.out.println(s2.aktiv);
  }
}
```

# Überschriebene Methoden

Zur Laufzeit wird die Klasse des Objekts festgestellt, auf das die Referenz gerade verweist. Beginnend mit dieser Klasse wird nun die Klassenhierarchie in Richtung der Superklassen abgesucht, bis die erste Methodendeklaration gefunden wird, die dieselbe Signatur (und damit zwangsläufig denselben Ergebnistyp) wie die spezifischste Methode hat.

Diese **überschreibt** alle weiter oben deklarierten Methoden mit derselben Signatur, und sie wird ausgeführt.

# Beispiel (1)

```
class X {
  void drucke() {
    System.out.println(“X“);
  }
}

class Y extends X {
  void drucke() {
    System.out.println(“Y“);
  }
}
```

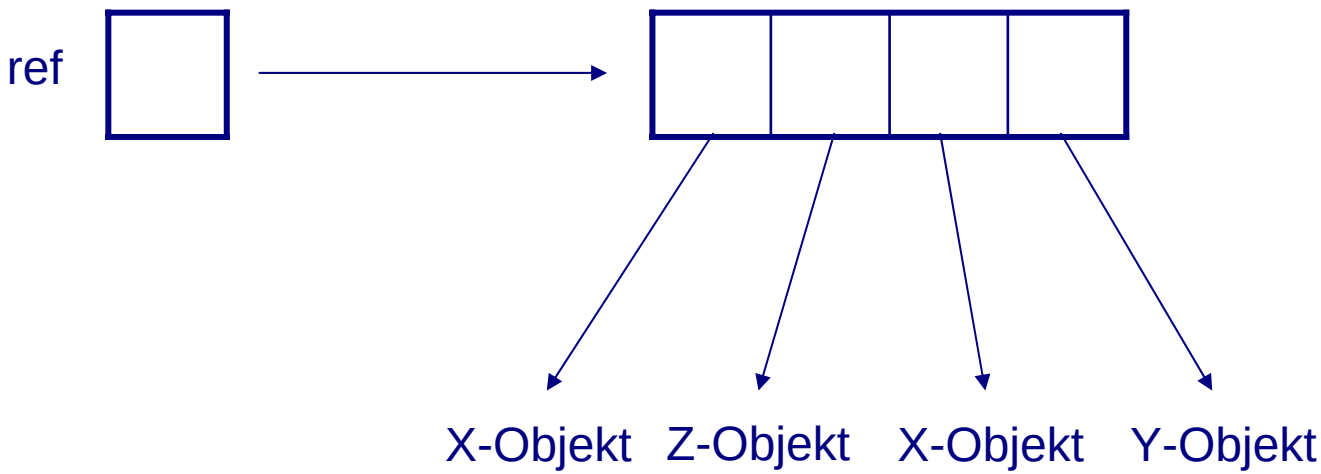
# Beispiel (2)

```
class Z extends Y {
  void drucke() {
    System.out.println(“Z“);
  }
}
class XYZ {
  public static void main(String[] args) {
    X[] ref = {new X(), new Z(), new X(), new Y()};
    for (int i=0, i<ref.length; i++)
      ref[i].drucke();
  }
}
```

Es werden X.drucke, Z.drucke, X.drucke und Y.drucke ausgeführt.

# Beispiel (3)

Veranschaulichung der Situation:



# Beispiel für die Klasse Kto

```
class Kto {
  String inhaber;
  long nummer;
  double stand, habenZins;
  Kto(String inhaber, long nummer, double stand,
      double habenZins) {
    this.inhaber = inhaber;
    this.nummer = nummer;
    this.stand = stand;
    this.habenZins = habenZins;
  }
  void info() {}
}
```

# Beispiel für die Klasse GiroKto (1)

```
class GiroKto extends Kto {
  private double sollZins, kreditLimit;
  GiroKto(String inhaber, long nummer, double stand,
          double habenZins, double sollZins,
          double kreditLimit) {
    super(inhaber, nummer, stand, habenZins);
    this.sollZins = sollZins;
    this.kreditLimit = kreditLimit;
  }
  GiroKto(String inhaber, long nummer, double stand,
          double habenZins, double sollZins) {
    this(inhaber, nummer, stand, habenZins, sollZins,
          2000.0);
  }
```

# Beispiel für die Klasse GiroKto (2)

```
  GiroKto(String inhaber, long nummer, double stand,
          double habenZins) {
    this(inhaber, nummer, stand, habenZins, 13.5, 2000.0);
  }
  GiroKto(String inhaber, long nummer, double stand) {
    this(inhaber, nummer, stand, 2.0, 13.5, 2000.0);
  }
  void info() {
  System.out.println("\nInhaber: " + inhaber
  + "\tKto-Nr: " + nummer + "\nKto-Stand: " + stand
  + " Euro" + "\nHabenzinsen: " + habenZins + " %"
  + "\tSollzinsen: " + sollZins + " %"
  + "\nLimit: " + kreditLimit + " Euro");
  }
// Weitere Methoden: einzahlen, abheben, etc.
}
```

# Die Klasse FestgeldKto (1)

```
class FestgeldKto extends Kto {
  private int restLaufzeit;
  private boolean zinsBesteuerung;
  FestgeldKto(String inhaber, long nummer, double stand,
              double habenZins, int restLaufzeit,
              boolean zinsBesteuerung) {
    super(inhaber, nummer, stand, habenZins);
    this.restLaufzeit = restLaufzeit;
    this.zinsBesteuerung = zinsBesteuerung;
  }
  FestgeldKto(String inhaber, long nummer, double stand,
              double habenZins, int restLaufzeit) {
```

# Die Klasse FestgeldKto (2)

```
    this(inhaber, nummer, stand, habenZins, restLaufzeit,
          true);
  }
  void info() {
    System.out.println("\nInhaber: " + inhaber
       + "\tKto-Nr: " + nummer
       + "\nKto-Stand: " + stand + " Euro"
       + "\nHabenzinsen: " + habenZins + " %"
       + "\tRestlaufzeit: " + restLaufzeit + " Monate\n"
       + ((zinsBesteuerung) ? "Zinsbesteurung" :  "keine
          Zinsbesteuerung"));
  }
// Weitere Methoden: auflösen, verlängern, etc.
}
```

# Anwendungsbeispiel (1)

```
class KtoTest {
 public static void main(Strings[] args) {
  Kto[] ktoFeld = new Kto[10000];
  ktoFeld[0] = new GiroKto(“S. Lucas“,301087,3020.15);
  ktoFeld[1] = new GiroKto(“G. Schiek“,306361,7812.64,0.3);
  ktoFeld[2] = new FestgeldKto(“F. Wild“,550001,8500.0,3.25,3);
  Kto kto;
  for (int i=0; i<ktoFeld.length; i++)
      if((kto = ktoFeld[i]) != null)
            kto.info();
```

# Anwendungsbeispiel (2)

```
  ktoFeld[0] =
        new FestgeldKto(“W.Müller“,551007,30000.0,3.35,1);
   for (int i=0; i<ktoFeld.length; i++)
      if ((kto = ktoFeld[i]) != null)
        kto.info();
  }
}
```

# Methodenaufrufe mittels super (1)

Wenn eine Methode mit dem Schlüsselwort **`super`** anstelle einer Objektreferenz aufgerufen wird, beginnt Java die Suche nach der auszuführenden Methode in der direkten Superklasse der Klasse, in der der Aufruf erfolgt, und sucht die Vererbungshierarchie in Richtung der Superklassen ab. Dies kann man sinnvoll einsetzen, um rekursive Aufrufe derselben Methode zu verhindern oder um Codeduplizierung durch Auslagerung in die Superklassen zu vermeiden.

Im `Kto`-Beispiel ist es zwecksmäßig, den leeren Rumpf in der `Kto.info`-Methode zu ersetzen und entsprechend die Methodendeklarationen in den Subklassen zu vereinfachen, wie im Folgenden dargestellt:

# Methodenaufrufe mittels super (2)

```
class Kto {
  void info() {
    System.out.println("\nInhaber: " + inhaber
        + "\tKto-Nr: " + nummer
        + "\nKto-Stand: " + stand + " Euro"
        + "\nHabenzinsen: " + habenZins + " %");
  }
  …
}
```

# Methodenaufrufe mittels super (3)

```
class GiroKto extends Kto {
  void info() {
    super.info();
    System.out.println(“\tSollzinsen: “ + sollZins + “ %“
        +“\nLimit: “ + kreditLimit + “ Euro“);
  }
  …
}
```

# Methodenaufrufe mittels super (4)

```
class FestgeldKto extends Kto {
  void info() {
    super.info();
    System.out.println("\tRestlaufzeit: " + restLaufzeit
        + " Monate\n"
        + ((zinsBesteuerung)? "Zinsbesteuerung" :
           "keine Zinsbesteuerung"));
  }
  …
}
```

# `final` Methoden (1)

Deklarationen von Instanz- und Klassenmethoden kann man mit dem Modifizierer **`final`** einleiten. Dies bewirkt, dass die Methode in Subklassen nicht mehr überschrieben werden kann.

Beispiel:

```
class Punkt {
  int xKoord, yKoord;
  final void verschiebe(int dx, int dy) {
    xKoord += dx;
    yKoord += dy;
  }
  …
}
```

# final Methoden (2)

```
class Farbpunkt extends Punkt {
  Color c;
  void verschiebe(Punkt p) {
    xKoord += p.xKoord;
    yKoord += p.yKoord;
  }
  void verschiebe(int dx, int dy) {…}    // Fehler
  …
}
```

# `final` Klassen

Neben Methoden können auch Klassen als **`final`** deklariert werden, indem man den Modifizierer **`final`** vor **`class`** setzt. Dies bedeutet, dass die Klasse „komplett“ ist und dass Spezialisierungen in Subklassen weder benötigt werden noch erwünscht sind.

Der Versuch, von einer **`final`**-Klasse eine Subklasse abzuleiten, ist ein Fehler. Weiterhin sind alle Methoden einer **`final`**-Klasse implizit **`final`**.

# Abstrakte Klassen

Andererseits ist es möglich, dass die Erzeugung von Instanzen einer Superklasse ausgeschlossen sein sollte, da mit ihr nur ein gemeinsames Konzept festgelegt wird, dessen verschiedene Ausprägungen in den Subklassen implementiert werden müssen. Ein Beispiel hierfür ist die Klasse **`Kto`**.

Die Erzeugung von Objekten kann man verhindern, indem man die Superklasse als **abstrakte Klasse** deklariert und ihrer Deklaration dazu den Modifzierer **`abstract`** voranstellt.

```
abstract class Kto {…}

  …

  ktoFeld[0] = new Kto();   // Fehler

  …
```