Die Superklasse Object

Object ist die Basisklasse aller Java-Klassen und wird automatisch geerbt. Sie darf daher nicht mit *extends* geerbt werden.

Einige wichtige Methoden der Klasse Object:

public final Class getClass()
→ bestimmt den Laufzeit-Typ eines Objektes

protected Object clone()
→ erzeugt ein neues Objekt als exakte Kopie

public String toString()
→ liefert die String-Repräsentation eines Objektes

public boolean equals(Object obj)
→ überprüft, ob es sich um das gleiche Objekt handelt

public int hashCode()
→ bestimmt den Hashwert eines Objektes

public final class getClass()

```
class WhichClass {
    public static void main(String[] args) {
        Rechteck R1 = new Quader();
        Rechteck R2 = new Rechteck();
        System.out.println(R1.getClass().getName());
        System.out.println(R2.getClass().getName());
        }
    }
```

Ausgabe: Quader
Rechteck

R1.getClass() und R2.getClass():
Die Methode getClass() hat als Ergebniswert ein Objekt vom Typ *Class*

getName() ist eine Methode der Klasse *Class* und liefert als Ergebniswert den Namen der Klasse als String
→ public String java.lang.Class.getName()

protected Object clone()

Eine Klasse, die die Methode clone() zur Verfügung stellen will, muss zwei Dinge tun:

1. Die Klasse muss das Interface *Cloneable* implementieren, um deutlich zu machen, dass ein Objekt dieser Klasse kopiert werden darf. Das Interface Cloneable befindet sich im Package java.lang. java.lang darf nicht importiert werden, da es automatisch immer vorhanden ist.

2. Die Methode clone() muss mit dem Schlüsselwort *public* überschrieben werden.

```
public class Beispiel implements Cloneable {
        . . .
        public Object clone() {
                return super.clone();
                }
        . . .
        }
```

Bemerkungen:

Object.clone() überprüft das Vorhandensein des Interfaces *Cloneable*, allokiert den benötigten Speicher und kopiert das Objekt bitweise

Shallow-Copy:
bei Objektattributen wird nur die Referenz kopiert, nicht das Objekt selbst

public String toString()

Liefert als Ergebniswert: Klassenname + Hashwert

Der Hashwert eines Objektes ist so etwas wie eine Prüfsumme des Objektes. Unterschiedliche Objekte sollten unterschiedliche Hashwerte haben. Zu Hashwerten später mehr.

Standardimplementierung:

```
public String toString() {
        return getClass().getName() + "@" +
                Integer.toHexString(hashCode());
        }
```

Beispiel:

```
System.out.println("new Rechteck().toString());        // Rechteck@923e30
System.out.println("new Rechteck().toString());        // Rechteck@130c19b

/* Da es sich um zwei unterschiedliche Objekte der Klasse Rechteck handelt,
   sind die beiden Hashwerte unterschiedlich. */
```

Werden mehr Informationen (z.B. Attribute) benötigt $\Rightarrow$ toString() überschreiben

public boolean equals(Object obj)

Zwei Variablen x und y vom Datentyp *int* werden als gleich betrachtet, wenn sie denselben Wert haben.

if (x == y) liefert true, wenn der Wert beider Variablen gleich ist.

Das Überprüfen zweier Objekte auf Gleichheit geht natürlich nicht mit dem Operator ==, denn dieser würde nur überprüfen, ob die Referenzvariablen denselben Wert haben, also auf dasselbe Objekt verweisen. Um zwei Objekte auf Gleichheit zu überprüfen, bietet sich die Methode *equals* an. Gleichheit bedeutet in diesem Zusammenhang inhaltliche Gleichheit, das heißt, dass jedes Attribut des einen Objektes denselben Wert hat wie das entsprechende Attribut des anderen Objektes.

Beispiele:

```
Beispiel 1:   String s1;
              s1 = new String("Hello World");
              String s2;
              s2 = new String("Hello World");
              if (s1.equals(s2)) . . .          // liefert true

Beispiel 2:   String init = "Hello World";
              StringBuffer sb1;
              sb1 = new StringBuffer(init);
              StringBuffer sb2;
              sb2 = new StringBuffer(init);
              if (sb1.equals(sb2)) . . .        // liefert false
```

Das Ergebnis von Beispiel 2 überrascht auf den ersten Blick ein wenig. Obwohl beide StringBuffer sb1 und sb2 denselben Text "Hello World" beinhalten, liefert die Methode *equals* den Wert *false*. Im Beispiel 1 mit den beiden Strings s1 und s2 liefert *equals* hingegen true.

Der Grund für dieses unterschiedliche Verhalten ist die Standard-Implementierung der Methode *equals* in der Klasse *Object*.

```
public boolean equals(Object obj) {
        return this == obj;
        }
```

In der Standardimplementierung werden nur Referenzen verglichen, keine Inhalte. Die Klasse *String* überschreibt sinnvollerweise die Methode *equals*, so dass dort Inhalte verglichen werden. Die Klasse *StringBuffer* überschreibt die Methode nicht, so dass dort beim Aufruf von *equals* die Standard-Implementierung Anwendung findet.

Die Konsequenz ist, dass jede Klasse, bei der der Inhalt der Objekte (Attribute) wesentlich ist, *equals* überschreiben sollte.

Vorgehensweise beim Überschreiben der Methode equals:

```
public boolean equals(Object obj) {
        // Triviale Abfragen
        if (this == obj)  return true;                          // Es ist dasselbe Objekt
        if (obj == null)  return false;                         // Null-Referenz
        if (obj.getClass() != getClass())  return false;    // Unterschiedlicher Datentyp

        // Vergleich aller Attribute
        }
```

Vergleich der Attribute

- Attribute von einfachen Datentypen
- Attribute, die Referenzen auf Objekte sind, die eine korrekte Implementierung von equals() haben
- Attribute, die Referenzen auf Objekte sind, die keine korrekte Implementierung von equals() haben
- alle übrigen Attribute

Beispiel: Einfache Datentypen

```
class Test1 {
        private int x;
        . . .
        public boolean equals(Object obj) {
                // Triviale Abfragen
                if (x != ((Test1) obj).x)  return false;
                //Weitere Vergleiche von Attributen
                return true;
                }
        . . .
        }
```

Beispiel: Referenzen auf Objekte, die equals() überschrieben haben

```
class Test2 {
        private String huhu;
        . . .
        public boolean equals(Object obj) {
                // Triviale Abfragen
                if (huhu == null) {
                        if ((((Test2) obj).huhu) != null)
                                return false;
                        }
                else {
                        if (!(huhu.equals(((Test2) obj).huhu))
                                return false;
                        }
                // Weitere Vergleiche von Attributen
                return true;
                }
        . . .
        }
```

Beispiel: Referenzen auf Objekte, die equals() nicht überschrieben haben

Man benötigt Sonderlösungen, Kreativität ist gefragt

z.B. bei StringBuffer
- → per toString()-Methode in String verwandeln
- → equals() von String nutzen

z.B. bei Arrays
- → Inhalt des Arrays elementweise vergleichen

public int hashCode()

Die Methode hashCode() liefert den Hashwert des Objektes. In der Standard-Implementierung in der Klasse *Object* wird als Hashwert die Speicheradresse der Objekt-Referenz zurückgegeben.

Wird in einer Klasse die Methode equals() überschrieben, muss auch die Methode hashCode() überschrieben werden. Der Grund ist, dass zwei Objekte, die nach der Methode equals() als gleich betrachtet werden, denselben Hashwert haben müssen (ansonsten wären sie nicht gleich). Umgekehrt gilt das natürlich auch: Wenn in einer Klasse die Methode hashCode() überschrieben wird, muss auch die Methode equals() überschrieben werden.

Eine wesentliche Eigenschaft eines Hashwertes:

> Über den Hashwert ist ein sehr schneller (indexbasierter) Zugriff auf gespeicherte Informationen möglich. Dazu mehr, wenn wir über das Collection-Framework reden.

Anleitung zur Implementierung von hashCode()

Idee:

→ Attribute werden entsprechend equals() berücksichtigt
→ je Attribut: Integer-Wert*Multiplikator = Hash
→ alle Hash werden addiert

```
public int hashCode() {
    int hc = 17;                    // willkürlicher Anfangswert
    int multiplikator = 59;         // nicht zu große Primzahl

    // hc = hc + multiplikator*hash_Attribut1 + multiplikator*hash_Attribut2 + …
    return hc;
    }
```

Folgendermaßen lassen sich die Einzel-Hashwerte bestimmen:

Datentyp des Attributs	**Zugeordnete Einzel-Hashwerte (Integer-Werte)**
boolean	if (Attribut == true) {hash=1;} else {hash=0;}
byte, char, short, int	hash = (int) Attribut;
long	zerlegen in zwei int → hash = Int1 + Int2;
float	In int-Wert umwandeln : Float.floatToIntBits(Attribut))
double	In long-Wert umwandeln : Double.doubleToLongBits(Attribut)) und anschließende Behandlung wie long
Referenz	Attribut.hashCode()

Anmerkung:

> float oder double: Sollte das Attribut den Wert 0 haben, ist der Einzel-Hashwert auch 0.
>
> Referenz: Ist die Referenz null, hat der Einzel-Hashwert den Wert 0.