Objektorientierte Konzepte in Java

Autor: Stephan Karrer

Abstrakte Datentypen und Kapselung

Klasse:	Kreis
Attribute (Daten):	Koordinate x Koordinate y Radius r
Methoden:	umfang() fläche() vergrößere()

Klasse:	Konto
Attribute (Daten):	Saldo s Limit k Letzte Buchung x
Methoden:	buche() setzeLimit() leseLimit() leseLetzteBuchung()

Typische Aufgabenstellung: Modellierung als Datentyp

Abstrakte Datentypen in Java abgebildet

```
public class Kreis       //in der Regel eigene Quellcode-Datei
{
     private double x,y,r;

     //expliziter Konstruktor
     public Kreis(double x_koor, double y_koor, double radius)
     {
          x = x_koor;
          y = y_koor;
          if ( radius < 0)
          {    r = 0;
          }
          else
          {    r = radius;
          }
     }

     public double umfang()
     {
          return (2 * r * 3.14);
     }

     public double fläche()
     {
          return (r * r * 3.14);
     }

     public void vergrößere(double faktor)
     {
          r = r * faktor;
          return;
     }
}
```

Aufrufender Code in Java

```
public class Kreistest            //Hauptprogramm, Aufrufer
{
   public static void main()
   {
         Kreis k1;
         .....k1, k2;
         k1 = new Kreis(0,0,5);
         ....
         k1.vergrößere(10);
         k2 = k1;
         .....
   }
}
```

- Daten sind bei Verwendung entsprechender Sichtbarkeitslevel nicht direkt zugreifbar
- Schnittstellen (Methoden) sind klar definiert
- Aus Sicht der einzelnen Klasse (abstrakter Datentyp) ist der Code robuster

Konsequenz: neue Sicht auf das Programm

- Ein Programm in einer OO-Sprache ist eine Codierung von Klassen (abstrakten Datentypen, die sowohl Methoden als auch Daten als Attribute besitzen).
- Zur Laufzeit des Programms existieren Objekte als Instanzen dieser Klassen, die gegenseitig ihre Methoden aufrufen und dadurch den Programmfluß realisieren.
- Das Hauptprogramm aus dem prozeduralen Ansatz reduziert sich auf ein Startprogramm, welches von der Laufzeitumgebung aufgerufen wird, die ersten Objekte instantiiert und die ersten Methoden aufruft, so daß der Programmfluß in Gang kommt.
  - **Das bedeutet: Das Startprogramm ist am unwichtigsten und kann zuletzt betrachtet werden, wesentlich ist das Design der Klassen (Verteilung des Codes) und deren Codierung.**

Klassen und Objekte in Java

```
public class Kreistest          //Hauptprogramm
{
   public static void main(String[] args)
   {
          ....
          Kreis k1, k2;
          k1 = new Kreis(0,0,5);
          ....
          k1.vergrößere(10);
          k2 = k1;
       System.out.println(k2.umfang());
          .....
   }
}
```

```
public class Kreis
{
   private double x,y,r;

   public double umfang()
   { … }

   public double fläche()
   { … }

   public void vergrößere
          (double faktor){ ... }
}
```

- Ein Java-Programm besteht aus der Codierung von Klassen
- Zugriff auf die sichtbaren Elemente erfolgt via: ***Bezeichner.Elementname***
  d.h. bzgl. der Instanz (Objekt): ***Objektreferenz.Elementname***
  und bzgl. der Klasse: ***(Paketname.)Klassenname.Elementname***

Sichtbarkeitslevel (Access Modifier) in Java

public	Für alle zugreifbar
private	Nur innerhalb der Klasse zugreifbar
protected	Innerhalb des Packets (und der Klasse) sowie durch abgeleitete (Kind-) Klassen zugreifbar
	Default: Innerhalb des Packets (und der Klasse) zugreifbar

- Sichtbarkeitslevel lassen sich sowohl für die Klasse als auch einzelne Attribute bzw. Methoden definieren
- Welche Pakete durch andere Pakete erreichbar sind liegt außerhalb der Java-Sphäre

Objekterzeugung

```
Kreis k2, k1 = new Kreis(0,0,5);   k2 = k1;

Date d; d = new Date(123564L);   String s = new String();
```

- Analog zu herkömmlichen Datentypen lassen sich auch Variable für Klassentypen deklarieren und Werte zuweisen.
- Die Instantiierung, sprich das Erzeugen eines Objekts zur Laufzeit, wird mit dem Schlüsselwort ***new*** deutlich gemacht
- Die Variable ist eine Referenz auf das Objekt und typ-gebunden
- **Vorsicht: Bei Objekten gilt in Java Referenzsemantik, bei den Basisdatentypen hingegen Wertsemantik !!**

Gleichheit und Kopien

```
int a=1; int b=a;  //wertmäßig gleich

Kreis k1 = new Kreis(0,0,2); Kreis k2 = k1; //Referenz ist gleich
Kreis k3 = new Kreis(0,0,2);  //k1 und k3 sind wertmäßig gleich
// k1 == k2 liefert true, k1 == k3 liefert false
```

Für alle Referenztypen (Klassen) und ihre Instanzen (Objekte) gilt:

- Bei einer Zuweisung wird nur die Referenz kopiert, das Objekt ist identisch
  - wir benötigen spezielle Funktion für Kopien: ***clone()***
- Als Eingabeparameter für eine Methode wird nur die Referenz übergeben, die Methode agiert auf dem ursprünglichen Objekt
- Prüfung auf Gleichheit vergleicht nur die Referenzen, nicht die Objekte
  - wir benötigen speziellen Test auf wertmäßige Gleichheit: ***equals()***

Die null-Referenz

```
String s = null;

Date d; if (d == null ) //...
```

- Für alle Referenzdatentypen (Klassen) ist der Standardwert für die Objektreferenz die **"leere"** Referenz: ***null***
- **null** kann jedem Objekt zugewiesen werden (genauer der Variablen) bzw. jeder Methode anstelle einer "echten" Objektreferenz übergeben werden
- Zur Laufzeit wirft der Interpreter beim Zugriff auf **null** eine ***NullPointerException***
- **null** als Argument oder Rückgabe sollte vermieden werden
  - **null** steht oft für einen Fehler, der unerkannt in eine Methode geht, die dafür eine Sonderbehandlung definieren sollte
  - Erlaubt eine Methode die Rückgabe **null** , so muss der Empfänger dies in der Regel testen
  - Die potentielle Rückgabe einer null sollte dokumentiert sein (API-Doc)

Konstruktoren

```
public class Kreis {
          // …
          public Kreis(double x_koor, double y_koor, double radius)
          {         x = x_koor;
                    y = y_koor;
                    if ( radius < 0)    { r = 0; }
                    else {     r = radius; }
          }
          // …
}
```

- Bei der Verwendung von new wird ein Konstruktor aufgerufen.
- Konstruktoren sind spezielle Methoden mit dem Namen der Klasse, die das Objekt initialisieren und eine Referenz auf das Objekt zurückliefern.
- Eine Klasse kann mehrere Konstruktoren haben (Überladen, Overloading), oft auch einen "leeren".
- Wird kein expliziter Konstruktor bereit gestellt, so werden durch den Compiler alle Datenattribute standardmäßig initialisiert und der sog. Standard-Konstruktor ("leere" Konstruktor) hinzugefügt.

Weiterführende Bemerkungen zu Konstruktoren

- Wenn wir den Standard-Konstruktor explizit schreiben, können wir ihn auch dokumentieren.
- Keine sonstige Methode sollte den Namen ihrer Klasse tragen.
  - Eclipse quittiert dies mit einer Warnung
- Es gibt gute Gründe für einen privaten Konstruktor, der die direkte Instantiierung von außen verhindert.
  - Die Klasse besitzt nur statische Methoden.
  - Die Klasse enthält nur Konstanten.
  - Singletons oder Fabriken bauen über statische Methoden Exemplare der eigenen Klasse oder Unterklasse.
  - Eine typsichere Aufzählung.

Überladen von Methoden

```
public class Kreis {

private double x,y,r;

          //mehrere explizite Konstruktoren
          public Kreis(double x_koor, double y_koor, double radius) {
                    x = x_koor; y = y_koor;
                    if ( radius < 0) { r = 0; }
                    else {    r = radius; }
          }

          public Kreis(double radius) {
                    x=0; y=0; r=(radius<0) ? 0 : radius;
          }

public void vergrößere(double faktor) {
                    r = r * faktor; return;
}

public void vergrößere() {
                    r = r * 2;
          }
// ...
}
```

Überladene Methoden

- Eine überladene Methode hat den gleichen Namen wie eine andere Methode. aber eine andere Argumentliste
  - Nicht relevant sind Rückgabetyp, Exception, Modifizierer!
- Beim Methodenaufruf muss immer klar sein, welche Methode aufgerufen wird. Verwechselungen durch (automatische) Typanpassungen müssen vermieden werden.
- Besonders eine Methode mit ***varags*** ist anfällig für Fehler, wenn sie überladen ist

Die Selbst-Referenz: this

```
public class Kreis {
private double x, y, r;
   public Kreis(double x, double y, double r)
   {        this.x = x;
            this.y = y;
            if ( r < 0)    { this.r = 0; } else { this.r = r; }
   }
   public Kreis(double r)
   {        this(0,0,r); }
}
```

- Innerhalb von Instanz-Methoden kann jederzeit die Instanz via ***this*** selbst referenziert werden.
- Ein anderer Konstruktor kann via ***this(...)*** aufgerufen werden.

Statische Klassenelemente

```
public static final PrintStream out   //aus java.lang.System

public static final double PI    //aus java.lang.Math

public static void main(String[] args){...}

public static int abs(int a) {...} //aus java.lang.Math
```

- Mit dem Schlüsselwort ***static*** werden Elemente markiert, die unabhängig von Objekten auf der Klasseneben existieren.
- Referenzierung erfolgt via: ***(Paketname.)Klassenname.Elementname***
- Statische Methoden können keine Instanzmethode oder -attribut ohne Objektreferenz nutzen

Statische Blöcke (Initialisierer)

```
class StaticTest {
   static int x, y;
   static { x = (int)(Math.random()*10);
            if (x < 5) {y = 1;} else { y = 2;}
            System.out.println("Static Initialiser done");
          }
   // …
}
```

- In komplexeren Situationen kann auch ein statische Block verwendet werden.
- Der statische Block wird beim Laden der Klasse als erstes abgearbeitet
  - Keine Übergabeargumente möglich
  - Dient üblicherweise der Initialisierung statischer Elemente

Statt Konstruktor statische Erzeugungsmethode anbieten

```
public class Singleton {

   private static Singleton s;

   private Singleton() {} //privater Konstruktor

   public static Singleton getInstance(){
            if (s == null) {s = new Singleton();}
            return s;
   }
}
```

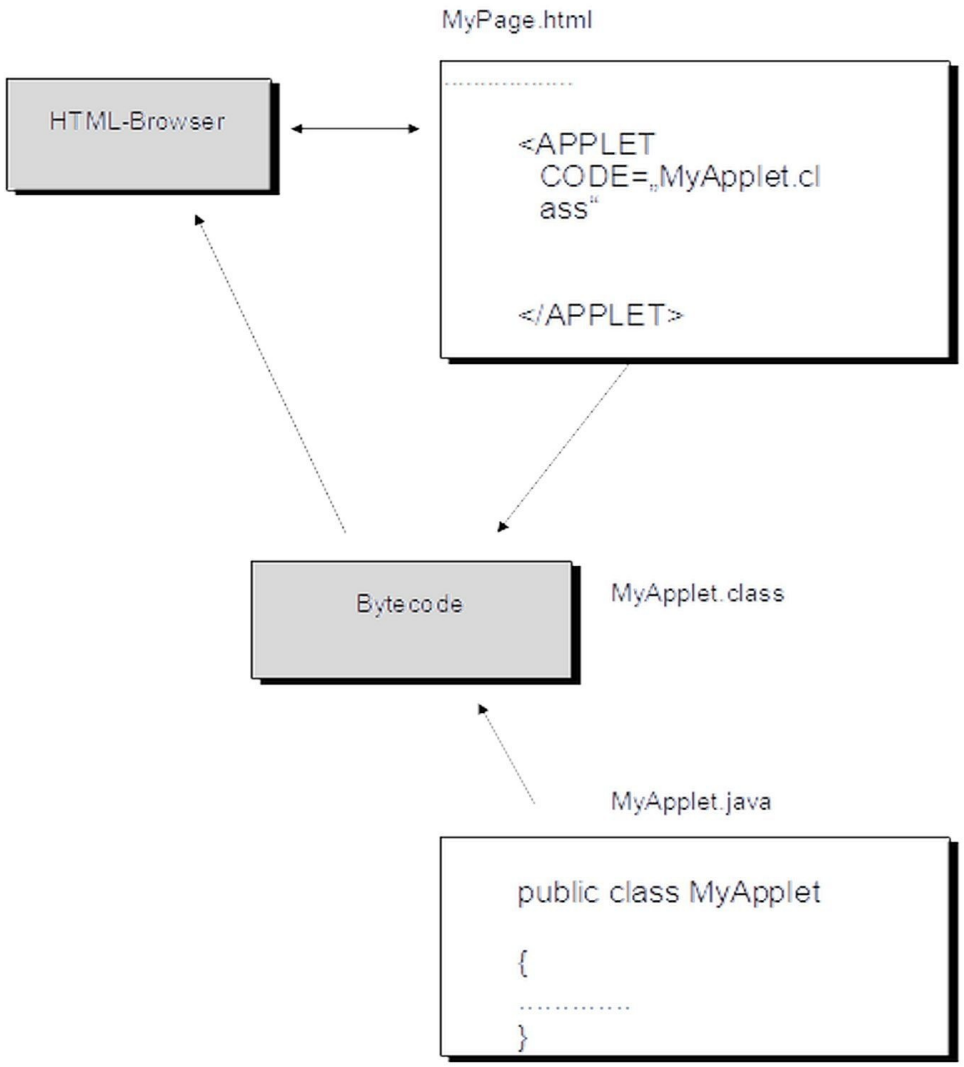
- Dies erlaubt Kontrolle über den Aufruf des Konstruktors.
- Typisch für Umsetzung des Factory- bzw. Singleton-Musters
- Vorsicht: obiger Code ist noch nicht Thread-safe !!

Destruktoren?

```
protected void finalize {   //...   }
```

- Da der Speicher automatisch aufgeräumt wird, entfallen explizite Destruktoren
- Der Garbage Collector kann nicht explizit aufgerufen werden, sondern nur ein Flag zur Signalisierung gesetzt werden: ***System.gc()***
- Aufräumarbeiten können optional je Objekt durch eine ***finalize***-Methode definiert werden. Diese wird durch den Garbage Collector gerufen, wenn der Speicherplatz des Objekts frei gegeben werden soll.
  - Zeitpunkt des Aufrufs ist nicht determiniert
  - Wird eher selten benutzt
  - ***Deprecated*** ab Java 9

Vererbung am Beispiel Java Applets (ab Java 9 deprecated)



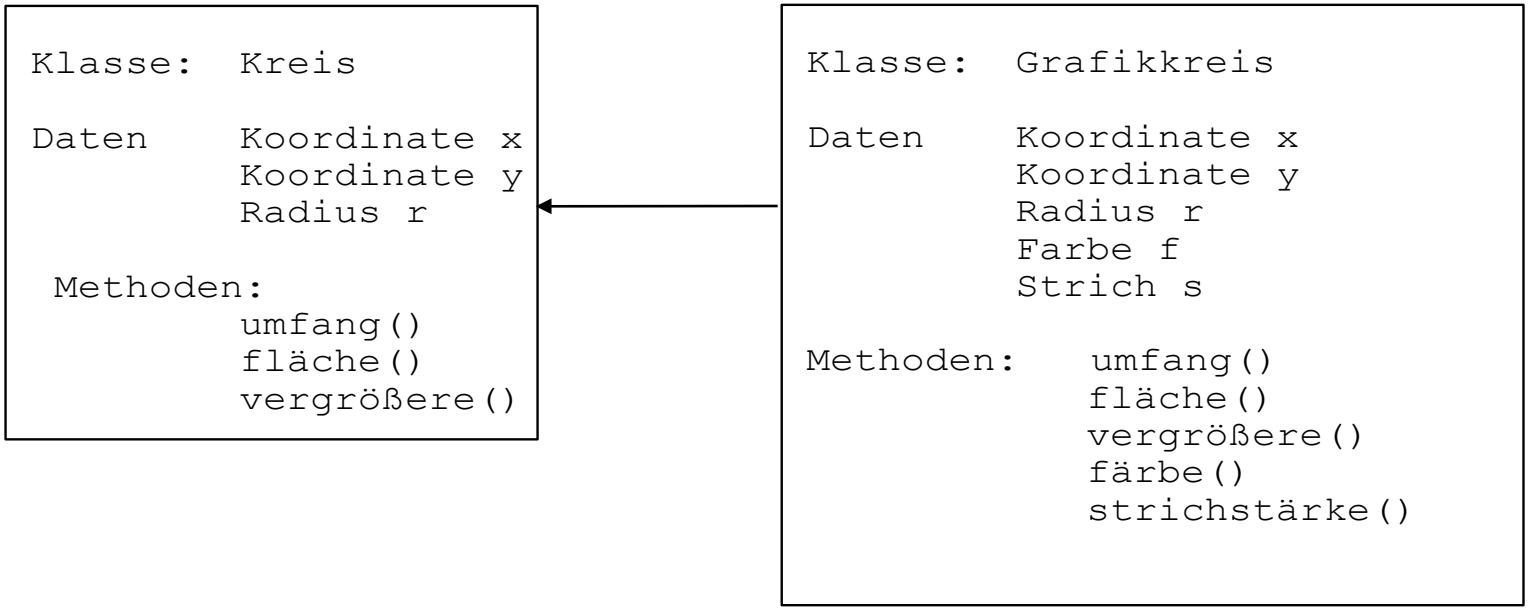
Aufbau eines Java Applets

```
import java.applet.*;
import java.awt.*;

public class HelloApplet extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("Java ist heiss!", 25, 50);
    }
}
```

Zum Test ausführbar via JDK-Tool AppletViewer
(ab Java 11 nicht mehr dabei)

Vererbung (inheritance)



Vererbung in Java

```
public class Kreis  //in der Regel eigene Quellcode-Datei
{       private double x,y,r;
        public Kreis(double x_koor, double y_koor, double radius)
        { .... }
        public double umfang()
        { .... }
        public double fläche()
        { .... }
        public void vergrößere(double faktor)
        { .... }
}
```

```
public class Grafikkreis extends Kreis
{       private int farbe, strich;
        public Grafikkreis(double x, double y, double r, int f, int s)
        {  super(x,y,r);    farbe = f;    strich = s;       }
        public void färbe()
        { .... }
        public void strichstärke()
        { .... }
}
```

Vererbung in OO-Sprachen

- Vorteil: Code (Daten und Methoden) muss nur einmal implementiert werden
- Ein Verfahren um in Kombination mit sauberer Klassenbildung Code wieder zu verwenden
- Ein Verfahren um gleichzeitig stabile Verwendbarkeit und dynamische Erweiterbarkeit zu realisieren
- Generelles Prinzip (Spezialisierung/Generalisierung) ist allgegenwärtig, speziell ist die Unterstützung durch Programmiersprache und Laufzeitsystem

Vererbung: Zugriff auf Elemente

```
class Eltern { String t="Eltern"; char k='E';
          void print() { System.out.println("Hier Eltern"); }
}

public class Kind extends Eltern {
         String s="Kind";  char a='K';

         void kprint(){ System.out.println("Hier Kind"); }

         public static void main(String[] args) {
                  Kind kind=new Kind();
                  kind.print(); kind.kprint();
                  System.out.println(kind.t+" : "+kind.k);
                  System.out.println(kind.s+" : "+kind.a);
         }
}
```

Auch die (sichtbaren) Attribute und Methoden der Elternklasse sind über die Kindklasse zugreifbar

Vererbung: Verdecken von Attributen

```
class Eltern { String t="Eltern"; char k='b'; }

public class Kind extends Eltern {
        String t="Kind";
        int k=2;

        public void print(){
        System.out.println(super.t+" : "+super.k);
        System.out.println(t+" : "+k);
        }

        public static void main(String[] args) {
        Kind kind=new Kind();
        kind.print();
        }
}
```

- Datenattribute der Elternklasse können verdeckt werden, hierfür genügt bereits die Verwendung des gleichen Bezeichners
- Mittels des Schlüsselworts ***super*** können die Attribute der Elternklasse zugegriffen werden.
  (***super*** kann nicht kaskadiert werden, um in der Vererbungshierarchie weiter nach oben zu gehen!)

Vererbung: Überschreiben und Überlagern

```
class Eltern {
        int f1() { return 1;      }
        int f2() { return 2;      }
        private void f3() { return; }
}

class Kind extends Eltern {
        int f1() { return(11 + super.f1()); }  //Überschreiben
        int f2(int i) { return 12; }       //Überladen
        void f3 () { return; }    //Definition
}
```

- Die überschriebene Methode ***f1()*** der Elternklasse kann via ***super.f1()*** aus der Kindklasse aufgerufen werden
- ***final*** verhindert das Überschreiben der Methode durch eine Kindklasse
- Die überschreibende Methode darf nicht weniger Zugriff gestatten (und auch keine ***Checked Exceptions*** werfen, sofern die überschriebene Methode dies nicht vorsieht) !
- Ab Java 6 kann die überschreibende Methode eine spezielleren Rückgabetyp benutzen (kovariantes Subtyping)

Polymorphismus

```
Object o1 = new String();        Object o2 = new Date();

String s = o2.toString()   //ruft die speziellere Methode auf

/* der Compiler prüft allerdings nur, ob für den Typ Object von o2
   die toString-Methode existiert */
```

- Ein spezialisiertes Objekt ist stets auch ein Objekt der allgemeinen Art
- Wo ein allgemeines Objekt erwartet wird, kann stets ein spezielleres verwendet werden.
- Es wird stets die „speziellere“, überschriebene Methode zur Laufzeit aufgerufen (der Compiler weiss davon nichts)
- Bei Verdeckung der Attribute oder bei verdeckten statischen Methoden wird nicht zur Laufzeit automatisch die „speziellere“ Variante der Kindklasse genutzt

Typanpassung bei Klassentypen

```
Object o1 = "Hallo";  Object o2 = "hallo";
boolean b = o1.equals(o2);  //ist für Compiler ok
String s = (String)o1;  //Cast für Compiler nötig
if (o1 instanceof String) {
       int len = ((String)o1).length();
   }
```

- Typanpassung bei Klassentypen bedeutet Überstimmen des Compiler, d.h. der Entwickler übernimmt die Verantwortung
- Bei der Typüberprüfung durch die Laufzeitumgebung resultieren tatsächliche Typverletzungen in einer ***ClassCastException***
- Mit dem ***instanceof***-Operator kann zur Laufzeit der Typ geprüft werden

Vererbung: Verketten der Konstruktoren

```
class Eltern { private int a, b;
        Eltern() { super(); }
        Eltern(int a, int b) { super(); //...
       }    //...
}

class Kind extends Eltern {   private int c;
        Kind() { this(1, 1, 1); }
        Kind(int a, int b, int c ){  super(a, b);
              //...
          }   //...
}
```

Es ist sicherzustellen, dass die Datenattribute der Elternklasse initialisiert werden

- Entweder wir rufen explizit einen Konstruktor der Elternklasse auf
- Oder der Compiler setzt automatisch den Aufruf des parameterlosen Konstruktors ein.

Aufgabe

```
class A {
  A()   {
    foo();
  }
  void foo() {
    System.out.println( "A" );
  }
}

class B extends A {
  void foo() {
    System.out.println( "B" );
  }
}
```

Frage: Was kommt bei

`new B().foo();`

auf den Bildschirm?

Vererbung in Java

- Was in der Kindklasse zur Verfügung steht, kann über die Sichtbarkeitslevel gesteuert werden: „partielles Erben“ ist möglich.

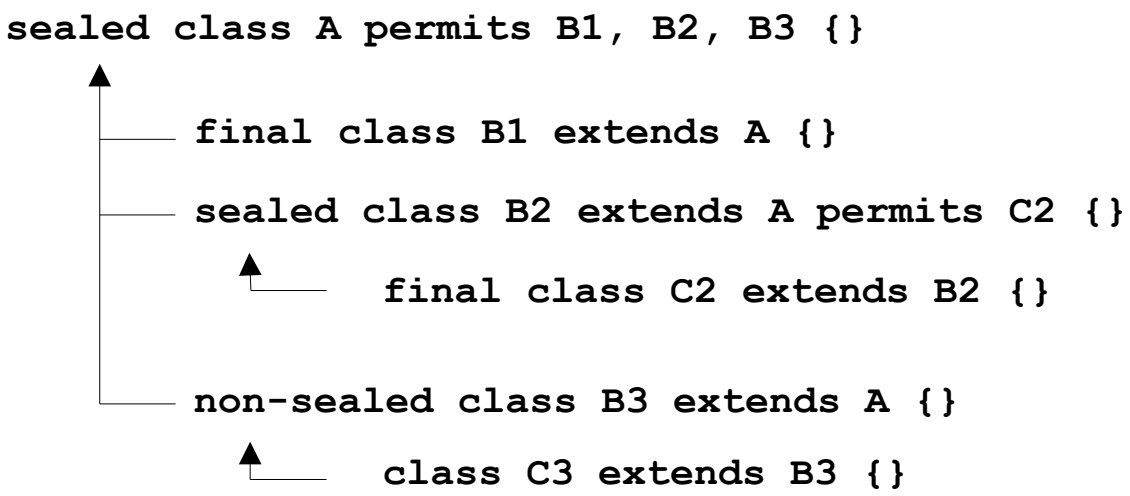
- Finale Klassen können nicht spezialisiert werden, d.h. es sind keine Kindklassen ableitbar.

```
public final class String extends Object
```

- Es gibt eine Basisklasse ***Object***, die Wurzel der Vererbungshierarchie bildet.
  - Alle Klassen erben grundlegende Elemente von ***Object***, sofern diese in der Klassenhierarchie nicht überschrieben wurden.

- Mehrfachvererbung ist nicht möglich (aber Implementierung mehrerer Schnittstellen).

Sealed Classes (ab Java 17, preview ab 15)



- Damit kann gezielt Vererbung eingeschränkt und Vererbungshierarchien fixiert werden
- Kindklassen von ***sealed*** Klassen müssen entweder ***final*** oder ***sealed*** oder ***non-sealed*** sein

Bedeutung von Schnittstellen

- Wie beim Vererben deutlich wurde, kann sich der nutzende Code (Aufrufer) auf die Funktionalität (Schnittstelle) der Elternklasse abstützen.

- Solange der nutzende Code nur die Schnittstelle benutzt, ist er von der konkreten Spezialisierung (Implementierung) der Kindklassen unabhängig.

- Durch die dynamische Entscheidung der Laufzeitumgebung wird auf der anderen Seite stets der „richtige“ Code (sprich die spezialisierte Form aufgerufen):
  z.B. ***toString***-Methode

- Die Bedeutung von Schnittstellen wird deshalb durch 2 spezielle Mechanismen in Java zusätzlich unterstützt:

  - Abstrakte Klassen
  - Interfaces

Abstrakte Klassen

```
public abstract class Number extends Object //...
       { //...
         public abstract int intValue();
         public abstract double doubleValue();
         //...
       }
```

- Von einer abstrakten Klasse können keine Objekte instantiiert werden
  - Eine abstrakte Klasse kann durchaus einen „expliziten“ Konstruktor haben
  - Ein Konstruktor kann nicht **abstract** sein
- In der Regel existiert mindestens eine abstrakte Methode in der Klasse, d.h. ohne Implementierung (sprich Rumpf)
  - Statische Methoden können nicht **abstract** sein
  - Abstrakte Methoden können (natürlich) nicht **private** sein

Kindklassen abstrakter Klassen

```
public abstract class Number extends Object //...
       { /* ... */ }

public final class Integer extends Number //...
       {  public int intValue(){ /* ... */  };
          /* ... */                                  }

public final class Double extends Number //...
```

- Eine „konkrete“ Kindklasse einer abstrakten Klasse muss alle abstrakten Methoden implementieren
- Ist das nicht der Fall, so muss die Kindklasse selbst wieder abstrakt sein
- Dadurch wird den Kindklassen eine bestimmte Schnittstelle vorgeschrieben

Verwendung abstrakter Klassen

```
class Beispiel {
      private Number n;
      Beispiel (Number n) { this.n = n; }
      int getValue () {  return n.intValue() * 2; }
 }

public class Beispieltest {
      public static void main(String[] arg){
         Beispiel b = new Beispiel(new Double(3.2));
         System.out.println(b.getValue());
         b = new Beispiel(new Integer(5));
         System.out.println(b.getValue());  }
 }
```

- Der nutzende Code kann sich auf den Typ der abstrakten Klasse und dessen Funktionalität abstützen bzw. beschränken
- Zur Laufzeit müssen die referenzierten Objekte natürlich „konkret“ sein

Schnittstellen (Interfaces)

```
interface Volumen {
        public abstract int getLänge();
        abstract int getBreite();
        int getHöhe();
}
```

- Ein Interface darf nur abstrakte Methoden und „Konstanten“ (static final Attribute) enthalten
  - Ist quasi abstrakte Klasse in Reinform
- Alle Methoden sind immer public und abstract
  - Die Schlüsselwörter ***abstract*** und ***public*** sind optional
  - Es gelten die üblichen Einschränkungen für abstrakte Methoden
  - Das Schlüsselwort ***abstract*** auf Ebene des Interface ist optional (Bsp: *abstract interface Volumen* …)

Implementierung von Schnittstellen (Interfaces)

```
class Car implements Volumen {
      /*  ... */
      public int getLänge(){ return 4; }
      public int getBreite(){ return 2; }
      public int getHöhe(){ return 2; }
      /* ... */  }
```

- Eine „konkrete“ Klasse muss alle Methoden des Interface implementieren
  - Ist die Implementierung nicht vollständig, so muss die Klasse **abstract** sein
- Um ein Interface zu erfüllen, genügt es auch , dass die Elternklasse das Interface implementiert, da die Implementierung vererbt wird
  - Das Schlüsselwort **implements** ist bei der Kindklasse dann optional
  - Die geerbten Methoden können natürlich in der Kindklasse überschrieben werden

Implementierung von mehreren Schnittstellen (Interfaces)

```
class Car implements Volumen, java.io.Serializable {
      /*  ... */
      public int getLänge(){ return 4; }
      public int getBreite(){ return 2; }
      public int getHöhe(){ return 2; }
      /* ... */  }
```

- Eine Klasse kann durchaus mehrere Interfaces implementieren
  - Ist quasi Ersatz für die Mehrfachvererbung
  - Sofern syntaktisch die gleiche Methode durch mehr als ein Interface vorgeschrieben wird, so erfüllt die Implementierung (syntaktisch) alle Anforderungen
- Ein Interface kann leer sein, z.B. **java.io.Serializable**
  - Dient als Markierung für den nutzenden Code

Schnittstellen erweitern Schnittstellen

```
interface Fläche {
      public abstract int getLänge();
      public abstract int getBreite();  }

interface Volumen extends Fläche, java.io.Serializable {
      public abstract    int getHöhe();  }
```

- Schnittstellen können eine oder mehrere Schnittstellen erweitern
  - Dies wird in der Klassenbibliothek durchaus verwendet, z.B.

    public interface java.util.List
    extends java.util.Collection

Verwendung von Schnittstellen bzw. implementierenden Klassen

```
import java.util.*;
class IteratorTest {      //Iterator ist ein Interface
      public static void main(String[] arg){
            ArrayList alist = new ArrayList();
            /* ... */
            Iterator iter = alist.iterator();
            while (iter.hasNext()){
                  System.out.println(iter.next());
            }     }     }
```

- Wie abstrakte Klassen können Interfaces als Typen verwendet werden
  - Der nutzende Code erwartet nur die Funktionalität des Interface, die sonstige tätsächliche Implementierung ist irrelevant

Verwendung von Schnittstellen als Konstanten-Pools

```
interface MeineKonstanten {
      public static final int konst1 = 1;
      public static final String konst2 = "Hallo";  }

class KonstantenTest implements MeineKonstanten {
      public static void main(String[] args){
            System.out.println(konst1);
            System.out.println(konst2);     }  }
```

- Neben Methoden-Deklarationen können auch „Konstanten“ definiert werden
  - Ist ab Java 5 durch die Bereitstellung von Aufzählungstypen (Enums) weitestgehend obsolet

Statische Methoden in Interfaces (ab Java 8)

```
public interface InterfaceWithStatic {

    int MINVALUE = 10;
    int MAXVALUE = 1_000_000;

    int getValue();

    public static boolean isValidValue(int i) {
        return (MINVALUE <= i) & (i <= MAXVALUE);
    }
}
```

- Statische Methoden in Interfaces sind auscodiert und müssen als solche gekennzeichnet werden.
  - Ist unkritische Erweiterung, da kein Vererbungsproblem
  - Werden stets anhand des Interface referenziert (keine Referenzierung anhand einer implementierenden Instanz möglich)

Factory-Method-Pattern mit statischen Methoden

```
// aus java.util

public interface Comparator<T> {
    //...

    static <T> Comparator<T>  comparingDouble (
                                    ToDoubleFunction<? super T> keyExtractor)
    { //…

    static <T extends Comparable<? super T>> Comparator<T>
                                     naturalOrder() { //...
```

- Das Factory-Method-Pattern kann nun via Interface umgesetzt werden!

Private Methoden in Interfaces (ab Java 9)

```
public interface InterfaceWithStatic {

    int MINVALUE = 10;
    int MAXVALUE = 1_000_000;

    int getValue();

    public static boolean isValidValue(int i) {
        return validate(i);
    }

    private static boolean validate(int i) {
        return (MINVALUE <= i) & (i <= MAXVALUE);
    }
}
```

- Konsequente Erweiterung, da ebenfalls kein Vererbungsproblem

Default Methoden in Interfaces (ab Java 8)

```
public interface InterfaceWithDefault {

    int MINVALUE = 10;
    int MAXVALUE = 1_000_000;

    int getValue();

    default int getRandomValue() {
        return getFixValue() + new Random().nextInt(MINVALUE);
    }

    private int getFixValue() {
        return 1;
    }
}
```

- Löst ein Problem bei Erweiterung eines vorhandenen Interface um neue Methoden: Bisher implementierende Klassen bekommen Default-Implementierungen via Vererbung verpasst.
- Die Default-Implementierung ist in der Regel nicht optimal und sollte überschrieben werden.

Jetzt existiert Mehrfachvererbung!

```
public class BrownEyes {
    public String getColor() {
        return "brown";
    }
}
```

```
public interface BlueEyes {
    default String getColor() {
        return "blue";
    }
}
```

```
public class WhatEyes extends BrownEyes implements BlueEyes{
    private String c = BlueEyes.super.getColor();
    public static void main(String[] args) {
        System.out.println( new WhatEyes().getColor());  //brown
        System.out.println( new WhatEyes().c);  //blue
}   }
```

- Ist kein Fehler: Elternklasse hat Vorrang
- Mittels entsprechendem super kann der jeweilige Elternteil angesprochen werden !

Mehrfachvererbung mit Konflikt

```
public interface GreyEyes {
    default String getColor() {
        return "grey";
    }
}
```

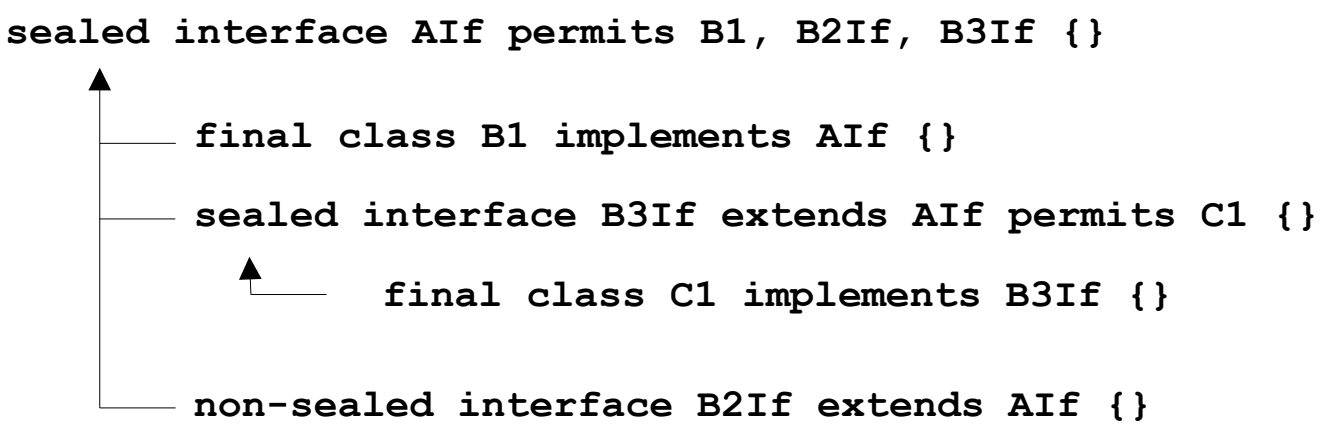
```
public interface BlueEyes {
    default String getColor() {
        return "blue";
    }
}
```

```
public class WhatEyes2 implements BlueEyes, GreyEyes{

    @Override
    public String getColor() {
        return BlueEyes.super.getColor();
    }
}
```

- Bei konkurrierenden Interfaces muß der Konflikt durch Überschreiben gelöst werden !

Sealed Interfaces (ab Java 17, preview ab 15)



- ***sealed*** ist auch für abstrakte Klassen und Interfaces anwendbar
- Für Interfaces wird dadurch sowohl Implementierung als auch Vererbung eingeschränkt.

Schnittstellen und abstrakte Klassen

- Der nutzende Code (Aufrufer) kann sich auf die Funktionalität der Schnittstelle abstützen.
- Solange der nutzende Code nur die Schnittstelle benutzt, ist er von der konkreten Implementierung unabhängig.
- Dies wird vor allem in Form von Schnittstellen im Java-Umfeld benutzt
  - In der Regel werden vorhandene Infrastrukturdienste durch die jeweiligen Hersteller mit Java-Schnittstellen ausgerüstet.
  - Durch die Standardisierung der Schnittstellen ist die Programmierung implementierungs-unabhängig.
  - Sehr viele Pakete der JEE bzw. sonstiger Frameworks nutzen in hohem Maße Schnittstellen und abstrakte Klassen.

Innere Klassen

```
class MyOuter {
        private String outer = "Outer";
        public String getName(){
                return outer; }
        private class MyInner {
                private String inner = "Inner";
                private String getName(){
                        return outer +":"+ inner;   }
        }       }
```

- Für spezielle Situationen besteht auch die Möglichkeit Klassen innerhalb von Klassen zu definieren.
  - Der Compiler erzeugt dabei 2 Class-Dateien: MyOuter.class und MyOuter$MyInner.class
- Die innere Klasse hat dabei Zugriff auf die Elemente der äusseren Klasse und umgekehrt (Sichtbarkeit spielt keine Rolle)

Nutzung innerer Klassen

```
class MyOuter {
        private String name = "Outer";
        public String getName(){
                MyInner in = new MyInner();
                return in.getName(); }
        private class MyInner {
                private String name = "Inner";
                private String getName(){
                  return MyOuter.this.name +":"+ name;   }
}   }
```

- Natürlich muss erstmal ein Objekt der inneren Klasse erzeugt werden, um deren Funktionalität zu nutzen
- Bei verdeckten Attributen kann in der inneren Klasse die Kombination von äußerem Klassennamen und this genutzt werden
- Eine solche innere Klasse kann keine statischen Elemente haben

Nutzung innerer Klassen von außen

```
class MyOuter {
        private String outer = "Outer";
        public String getName(){
                return outer; }
        class MyInner {
                private String inner = "Inner";
                String getName(){
                        return outer +":"+ inner;   }   }   }

class InnerTest{
        public static void main(String[] args){
                MyOuter mo = new MyOuter();
                MyOuter.MyInner mi1 = mo.new MyInner();
                MyOuter.MyInner mi2 = new MyOuter().new MyInner();
                mi1.getName();   }   }
```

- Zugriff ausserhalb der äusseren Klasse ist je nach Sichtbarkeit optional möglich
- Das Objekt der inneren Klasse kann nur anhand eines Objekts der äusseren Klasse erzeugt werden

Anonyme innere Klassen

```
public interface FlächeBerechnen {
double fläche ( Object o);
}


public class Kreis {
   //...
   FlächeBerechnen fl = new FlächeBerechnen()   //Innere Klasse
   {
        double fläche(Object o)
        { Kreis k = (Kreis) o;
          return (k.r * k.r * 3.14); }
   };
   //...
}
```

- Sofern die innere Klasse von einer vorhandenen Klasse abgeleitet wird oder ein Interface implementiert, kann sie auch anonym erstellt werden.
- Da sie keinen eigenen Namen besitzt, ist das nur bei einmaliger Verwendung sinnvoll.

Statische „innere Klassen“

- Hat eine innere Klasse keinen Bezug zur äußeren Klasse, kann diese statisch sein.
- Eine „echte“ innere Klasse existiert nur dann, wenn es auch ein Exemplar der äußeren Klasse gibt.
- Eine innere Schnittstelle ist immer automatisch öffentlich und statisch, deshalb können die Modifizierer `public static` entfallen.

Pakete

```
package anwendung1;
public class Programm1 { ... }
```

```
import anwendung1.Programm1;
import java.lang.*; //ist implizit immer gesetzt
public class TestAnwendung { ... }
```

- Ein Paket ist eine logische Gruppe von Klassen
  - wird üblicherweise auf eine entsprechende Verzeichnisstruktur abgebildet
  - Namenskonvention: Kleinschreibung ohne Sonderzeichen
- Fehlt die ***package***-Anweisung, so liegt die Klasse im sog. default-Paket.
- Durch Paketkonzept werden Namensräume etabliert
- Sichtbarkeitslevel beziehen sich auch auf Paketebene

Importieren von Namensräumen

```
import java.lang.*; //ist implizit immer gesetzt
import anwendung1.Programm1;  //die Klasse Programm1
import java.util.*;  //alle Klassen im Paket java.util
```

- Das default-Paket und ***java.lang*** sind stets eingebunden
- Bei Namenskonflikten müssen die voll-qualifizierten Bezeichner verwendet werden
- Der Compiler und der Klassenlader suchen die Pakete/Klassen der JSE anhand der System-Property ***sun.boot.class.path*** (ab Version 1.2)
- Für eigene Pakete muss üblicherweise die CLASSPATH-Variable entsprechend gesetzt werden
- Die Pakete liegen üblicherweise einschließlich Verzeichnisstruktur als ***jar***-Archive vor

Statischer Import

```
import static java.lang.System.out;
import static javax.swing.JOptionPane.showInputDialog;
import static java.lang.Integer.parseInt;
import static java.lang.Math.max;

class StaticImport
{
    public static void main( String[] args ) {
        int i = parseInt( showInputDialog( "First number" ) );
        int j = parseInt( showInputDialog( "Second number" ) );
        out.printf( "%d ist greater or equal.%n", max( i, j ) );
    }
}
```

Beim statischen Import werden nur bestimmte statische Elemente namenstechnisch eingebunden