



9. Vererbung

ARINKO[®]

- Bei der Erstellung einer neuen Klasse, kann man die Eigenschaften und das Verhalten einer bestehenden Klasse nutzen.
- Die neue Klasse wird dazu von der bestehenden Klasse abgeleitet und besteht dann aus zwei Anteilen:
 1. ein Anteil, der von der abgeleiteten Klasse stammt und
 2. ein Anteil, der neu hinzukommt
- Die bestehende Klasse bleibt unverändert und hat auch keine Kenntnis über von ihr abgeleitete Klassen.
- Die neue Klasse führt natürlich einen neuen Typ ins Typsystem ein, da sie abgeleitet ist, ist dieser neue Typ zusätzlich noch ein "erweiterter" Typ bezüglich des bereits bestehenden.
- Dieses Phänomen ist sehr wichtig für die Typprüfung.

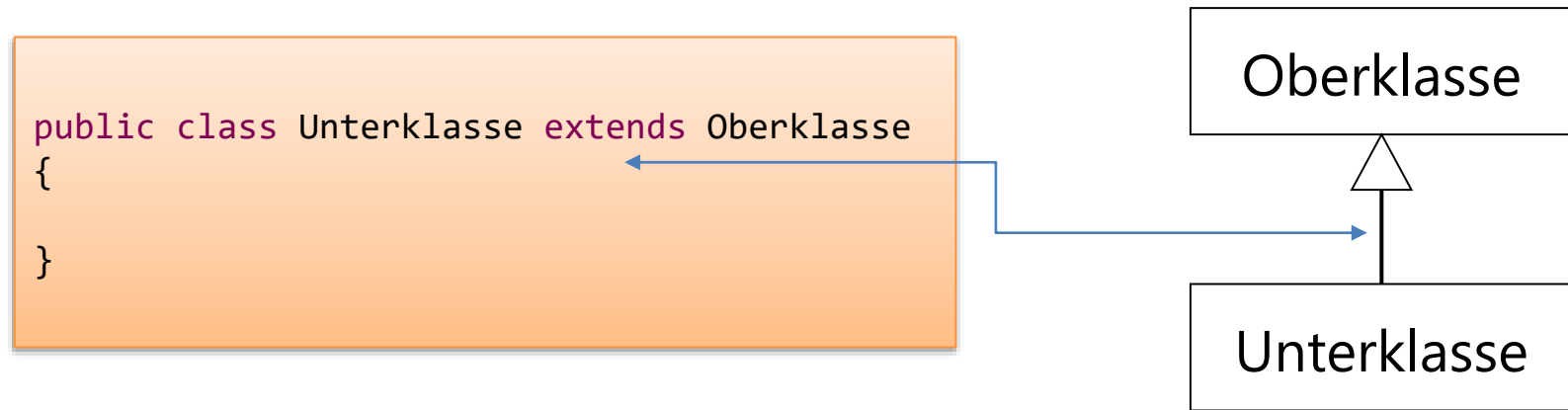
- Eine Ableitung zwischen Klassen wird syntaktisch so durchgeführt, dass bei der Deklaration der neuen Klasse das Schlüsselwort `extends` im Klassenkopf auftaucht:

```
public class Unterklasse extends Oberklasse  
{  
  
}
```

lies: "leitet ab von..."

- In Java hat man sich entschieden, "extends" (dt. "erweitert") zu verwenden. Manche Programmiersprachen benutzen "inherits" oder oft nur einen Doppelpunkt, z.B. C#
`public class Unterklasse : Oberklasse { ... }`
- Weiterhin ist hinter `extends` nur eine Klasse erlaubt, d.h. in Java existiert hier die sog. Einfachvererbung (Ggs. C++, dort ist Mehrfachvererbung erlaubt).
- Die abgeleitete Klasse ist *eine* **Unterklasse** (es kann mehrere geben).
- Die existierende Klasse ist *die* **Oberklasse** (es kann jeweils nur eine geben).

- Die UML benutzt einen gerichteten hohlen Pfeil, der von der Unterklasse auf die Oberklasse zeigt. Lesart ist somit in Pfeilrichtung "leitet ab von".



- Der Pfeilstrich muss eine durchgezogene Linie sein. Ist sie gestrichelt, bedeutet dies etwas anderes (Interface-Implementierung).

- Hier gibt es viele Synonyme:

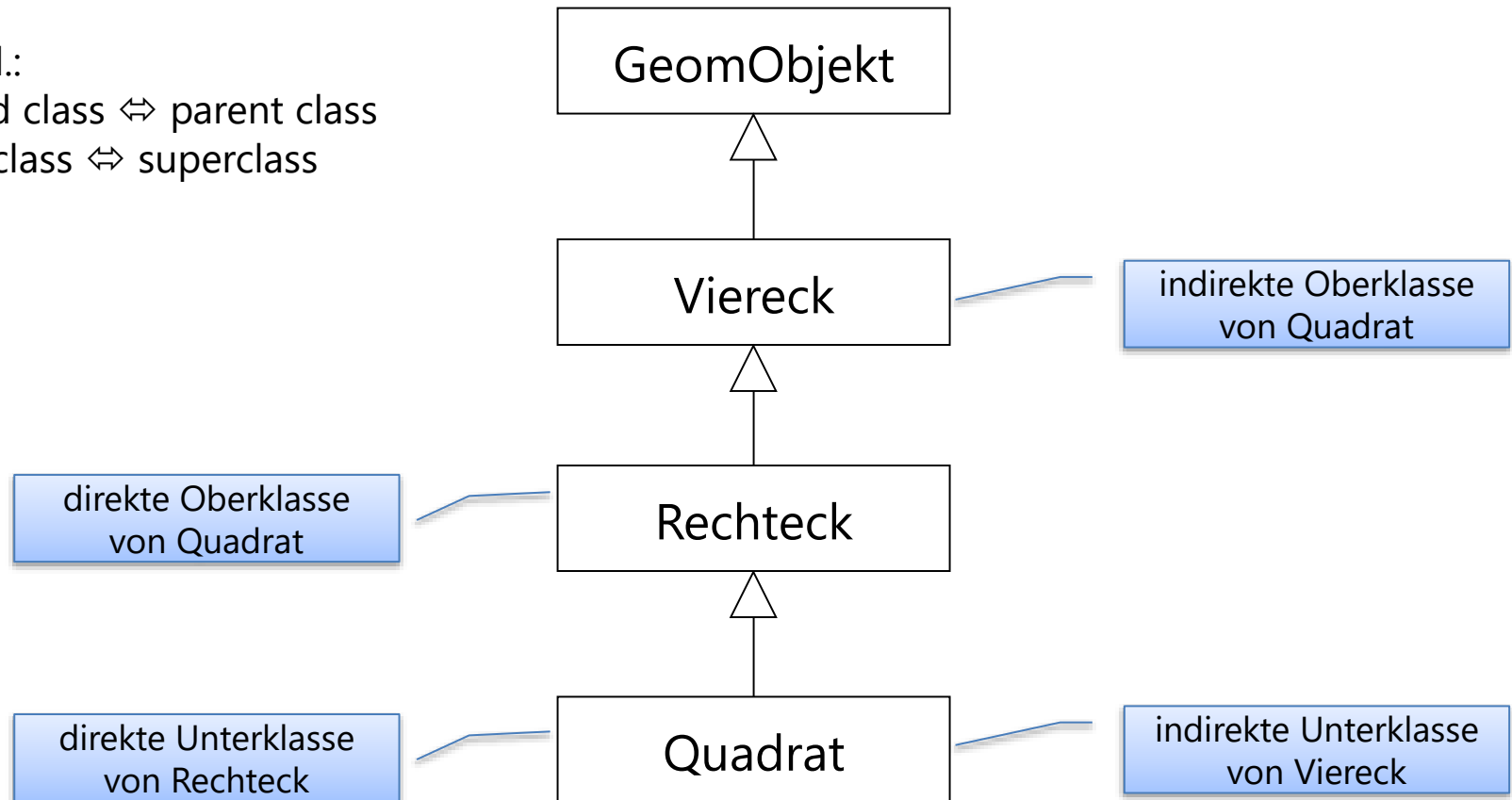
Unterklasse \Leftrightarrow Oberklasse

Subklasse \Leftrightarrow Superklasse

engl.:

child class \Leftrightarrow parent class

subclass \Leftrightarrow superclass



- Die Unterklasse "erbt" alle Elemente der Oberklasse.
- `public`-Elemente sind zugreifbar, als ob sie direkt in der Unterklasse selbst vereinbart wären.
- `private`-Elemente sind zwar vorhanden, aber nicht sichtbar/zugreifbar. (D.h. `private` Attribute "kleben" irgendwie an der Klasse, man kann sie aber nicht direkt modifizieren)

```
public class GeometrischesObjekt
{
    private Color linienfarbe;
    private Color fuellfarbe;
```

```
public class Kreis extends GeometrischesObjekt
{
```

Ein Kreis besitzt
Linien- und Füllfarbe,
die Attribute sind aber
nicht sichtbar

- Konstruktoren werden **nicht vererbt**. Jede Klasse braucht einen eigenen Satz von Konstruktoren.
- Konstruktoren können aber verkettet werden.
- Das heißt, dass im Konstruktor einer Unterklasse der Konstruktor der Oberklasse aufgerufen werden kann (genauer: muss), damit dieser die Bestandteile der Oberklasse initialisieren kann.

```
public class GeometrischesObjekt
{
    public GeometrischesObjekt(Color linienfarbe, Color fuellfarbe)
    {
        this.linienfarbe = linienfarbe;
        this.fuellfarbe = fuellfarbe;
    }
}
```

Kreis muss einen eigenen
Satz geeigneter
Konstruktoren bereitstellen

```
public class Kreis extends GeometrischesObjekt
{
    public Kreis(Point mittelpkt, int radius, Color linienfarbe, Color fuellfarbe)
    {
        super(linienfarbe, fuellfarbe);
        ...
    }
}
```

Ein Konstruktor der
Oberklasse kann zur
Unterstützung aufgerufen
werden

- public Elemente werden vererbt:

```
public class GeometrischesObjekt
{
    ...
    public Color getFuellfarbe()
    {
        return fuellfarbe;
    }
}
```

getFuellfarbe muss hier nicht erwähnt werden

```
public class Kreis extends GeometrischesObjekt
{
    ...
}
```

- Nutzung von außerhalb:

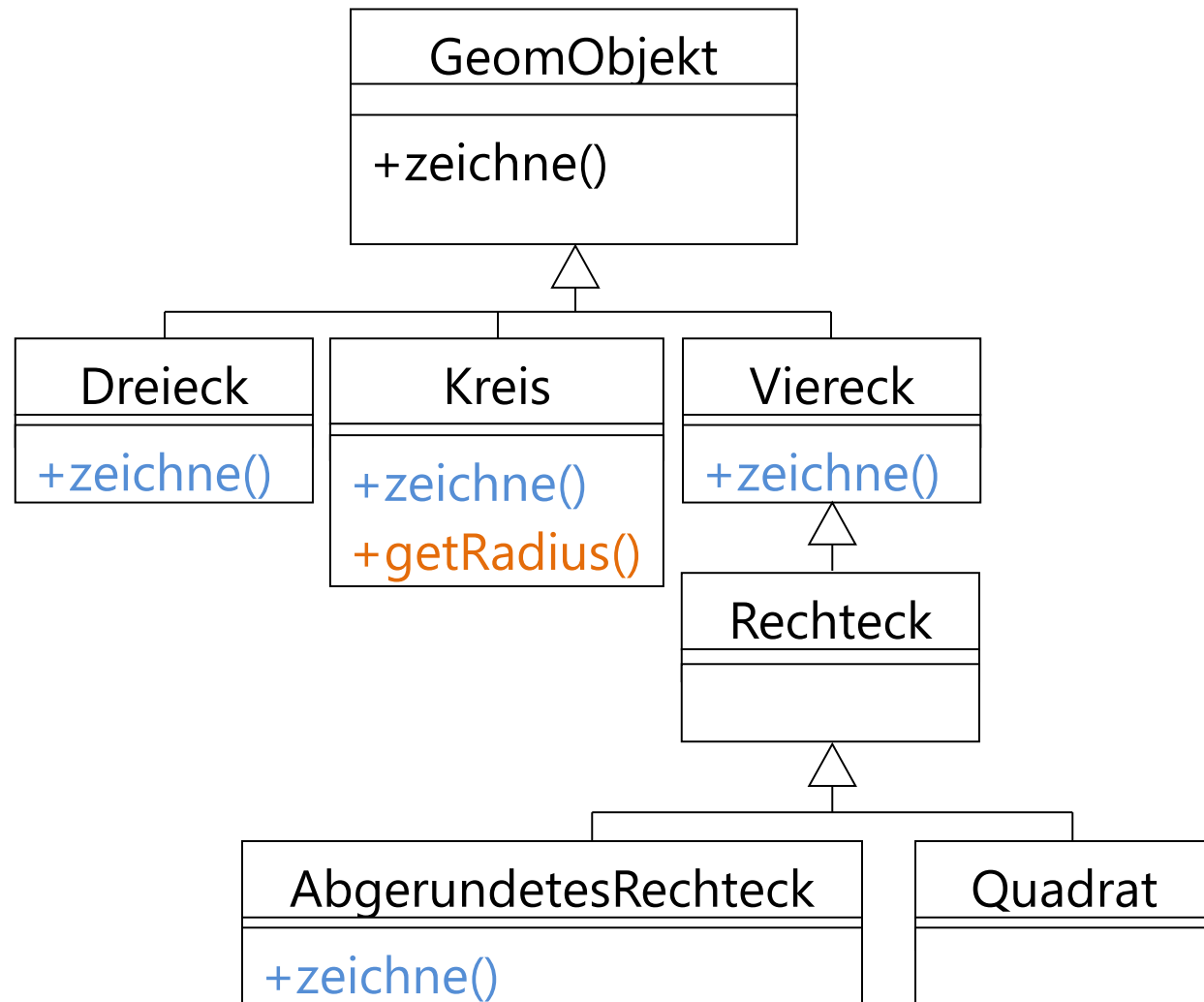
```
Kreis k = new Kreis(new Point(50,50), 10, Color.BLACK, Color.CYAN);
System.out.println(k.getFuellfarbe());
```

...ist aber an der Variable vom Typ Kreis vorhanden

- Jede Klasse ist direkte oder indirekte Unterklasse der Klasse Object.
- Natürlich mit Ausnahme der Klasse Object selbst, diese hat keine weiteren Oberklassen.
- Somit ist Object die Wurzel der Ableitungshierarchie.
- Wenn in der Klassendeklaration keine Ableitung definiert wurde, ergänzt der Compiler implizit `extends Object`.
- Somit ist garantiert, dass jede Klasse direkt oder indirekt von der Klasse Object erbt.
- In Object sind einige (zentrale) Eigenschaften hinterlegt, die jedes Java-Objekt können soll.
- Dazu später mehr.

- Abgeleitete Klassen haben bei Vererbung drei verschiedene Möglichkeiten, mit Elementen aus der Oberklasse umzugehen.
 1. Erweitern – es werden neue Elemente eingeführt, die es bislang in der Oberklasse noch nicht gab
 2. Überschreiben (engl. "overriding" [nicht overwriting!]) – geerbte Elemente werden verdeckt und durch ein eigenes Element ersetzt.
Eigentlich nur sinnvoll für Methoden.
 3. Vorgabe implementieren – es können in der Oberklasse gewisse Versprechungen gegenüber zukünftigen Nutzern von Unterklassen gemacht werden, die von der Unterklasse implementiert werden müssen.
Darauf kommen wir im Zusammenhang mit abstrakten Klassen und Interfaces zurück.
Genaugenommen ist dies aber nur ein Spezialfall von 2.)

- überschreiben
- erweitern



- Neue Elemente werden einfach in die Klasse geschrieben:

```
public class Kreis extends GeometrischesObjekt
{
    private Point mittelpunkt;
    private int radius;

    public int getRadius()
    {
        return radius;
    }
    ...
}
```

- Geerbte Methoden können (teilweise) mit einer neuen Implementierung versehen werden.
- Das nennt man dann "method overriding" (Überschreiben einer Methode).
- Die neue Methode muss in der Signatur (Name + Parameterliste) und Rückgabewert mit der Methode aus der Oberklasse übereinstimmen.
- Das Überschreiben verdeckt das geerbte Element, man kann in der überschreibenden Methode aber mit dem Schlüsselwort `super` auf das zugedekte Element zugreifen.
- Wichtig: `super` ist ein rein objektorientiertes Konzept und somit nicht in statischen Methoden verfügbar und es geht nur eine Etage, d.h. `super.super` ist nicht erlaubt.

```
public class Kreis extends GeometrischesObjekt
{
    public void zeichne()
    {
        // hier was eigenes
        super.zeichne();
        // und hier auch
    }
}
```

```
public class GeometrischesObjekt
{
    public void zeichne()
    {
    }
}
```

overriding

optionaler Zugriff auf die
Implementierung der Oberklasse

- Methoden können mit dem Modifizierer `final` versehen werden.
- Im Gegensatz zu Attributen ("letzte Belegung") bedeutet es hier "letzte Implementierung".
- `final` Methoden dürfen in Unterklassen nicht überschrieben werden.
- Auch Klassen dürfen den Modifizierer tragen.
- Dort bedeutet es, dass von der Klasse keine Ableitung erlaubt ist.

```
public class GeometrischesObjekt
{
    public final Color getFuellfarbe()
    {
        return fuellfarbe;
    }
    ...
}
```

- Java 17 führt die Möglichkeit ein, Vererbung etwas abgeschotteter zu betreiben.
- Klassen können gegen freie Vererbung versiegelt („sealed“) werden.
- Man definiert dann eine Liste von erlaubten Unterklassen („permits“).
- Die Unterklassen müssen dann entweder
 1. `sealed` (regeln ihrerseits wieder die Vererbung)
 2. `non-sealed` (erlauben weitere Vererbung) oder
 3. `final` (erlauben keine Vererbung) sein.

```
public sealed class GeometrischesObjekt permits Kreis, Rechteck  
{...}
```

```
public non-sealed class Kreis extends GeometrischesObjekt  
{...}
```

```
public final class Rechteck extends GeometrischesObjekt  
{...}
```

- Da Vererbung i.d.R. von der großen Vielfalt und Freiheit lebt, ist stark zu überlegen, ob und wann man dieses Feature nutzt.

- Nochmals: Konstruktoren werden nicht vererbt.
- Jede Unterklasse braucht ihren eigenen Satz von Konstruktoren.
- Ein Konstruktor der Oberklasse muss dennoch bei der Objekterzeugung aufgerufen werden, damit dieser die Möglichkeit hat, die Bestandteile der Oberklasse sauber zu initialisieren.
- Der Aufruf des Konstruktors der Oberklasse erfolgt entweder explizit oder implizit.
- Expliziter Aufruf mit

```
super( parameterliste );
```

- Somit hat man zwei Möglichkeiten in einem Konstruktor:
 1. Aufruf eines anderen Konstruktors der eigenen Klasse → `this()`
 2. Aufruf eines Konstruktors der Oberklasse → `super()`
- Die jeweils gewählte Möglichkeit muss die erste Zeile in einem Konstruktor sein!
- Impliziter Aufruf: ist in einem Konstruktor als erste Zeile keine der beiden Möglichkeiten codiert, ergänzt der Compiler den Aufruf an den parameterlosen Konstruktor der Oberklasse:

```
super();
```



```
public class Kreis extends GeometrischesObjekt
{
    public Kreis(Point mittelpunkt, int radius, Color linienfarbe, Color fuellfarbe)
    {
        super(linienfarbe, fuellfarbe);
        this.mittelpunkt = mittelpunkt;
        this.radius = radius;
    }

    public Kreis()
    {
        this(new Point(0,0), 0, Color.BLACK, Color.WHITE);
    }
    ...
}
```

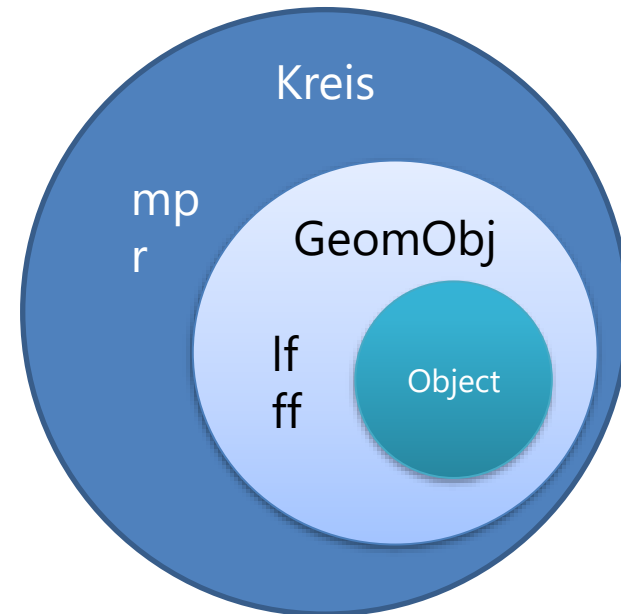
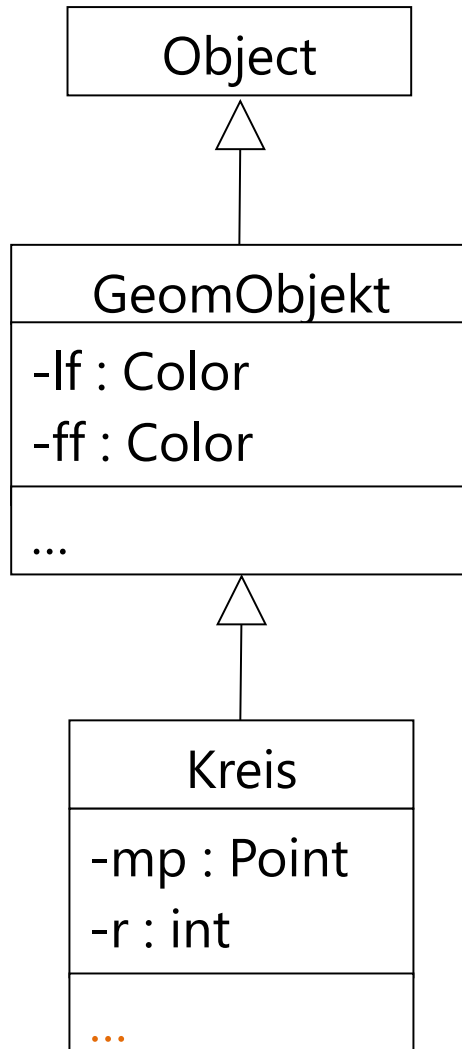
- Wo hat hier unser Programmierer einen Bug versteckt?

```
public class Oberklasse
{
    private int value;

    public Oberklasse(int a)
    {
        this.value = a;
    }
}
```

```
public class Unterklasse extends Oberklasse
{
}
```

- Zuerst wird Speicher für das gesamte Objekt (inkl. aller Attribute der Oberklassen) reserviert.
- Dann werden alle Attribute mit den Standardwerten (`0`, `null`, `false`) belegt.
- Danach wird der entsprechende Konstruktor aufgerufen.
- a) ist die erste Anweisung `this()`, dann Aufruf des entsprechenden Konstruktors der eigenen Klasse und weiter bei a)
- b) ist die erste Anweisung `super()`, dann Aufruf des entsprechenden Konstruktors der Oberklasse – dort beginnt es wieder bei a) bis wir in der Klasse `Object` angekommen sind.
- Je Ebene gilt:
 - Bevor die zweite Anweisung im Konstruktor ausgeführt wird, werden die Initialisierer der Attribute abgearbeitet.
 - Danach werden Initialisierungsblöcke von oben nach unten ausgeführt.
 - Danach springe eine Ebene zurück





10. Typkompatibilität und Polymorphismus

ARINKO[®]

- Bislang galt: rechts und links des Zuweisungsoperators müssen gleiche Typen stehen:
`Typ t = new Typ();`
- Da durch die Ableitung/Vererbung Klassen in engerem Zusammenhalt stehen, als beliebige Typen, weichen wir diese Regel auf.
- In einer Referenzvariablen können Referenzen abgelegt werden, deren Typ zum Typ der Variablen **kompatibel** ist.
- kompatibel: alle von einer Klasse direkt oder indirekt abgeleiteten Klassen sind zu dieser Oberklasse kompatibel.
- Beispiel:

```
GeometrischesObjekt go = new Kreis();
```

- Oder allgemeiner:

```
AllgemeinererTyp at = new SpeziellererTyp();
```

Was geht nicht?

- An Objekten, die mit einem allgemeineren Typ ausgezeichnet sind, dürfen keine spezielleren Elemente abgefragt werden.
- Der Compiler achtet nur auf den Typ!

Was geht?

- Beliebige kompatible Typen kommen als Belegung einer Variablen in Frage, das ist v.a. in Arrays oder für Parameter interessant.

```
GeometrischesObjekt go = new Kreis(); // kompatibel
GeometrischesObjekt[] gos = new GeometrischesObjekt[2];
gos[1] = new Kreis(); // kompatibel

go.getRadius(); // error: cannot find symbol
```

```
private static void zeichneBild(GeometrischesObjekt[] array)
{
    for (GeometrischesObjekt geometrischesObjekt : array)
    {
        geometrischesObjekt.zeichne();
    }
}
```

- Mit dem Operator `instanceof` kann man Objekte befragen, ob es zu einem gegebenen Typ kompatibel ist, Typgleichheit eingeschlossen.
- Das Resultat ist `boolean`, der linke Operand eine Referenz, der rechte Operand ein Typname.

```
GeometrischesObjekt go = new Kreis(); // kompatibel

if (go instanceof Kreis)
{
    //...
}
```

- Der Compiler lässt bei `instanceof` nur plausible Möglichkeiten zu.
- Das hier geht nicht:

```
if (go instanceof String)
{...}
```


- Die Zuweisung zu einem allgemeineren Typ mit Hilfe der Typkompatibilität stellt bereits eine Typumwandlung dar.
- Diese kann aber stillschweigend (d.h. implizit) erfolgen, da der Compiler sicherstellt, dass diese Umwandlung "safe" ist.
- Wenn es wichtig ist, den umgekehrten Weg zu gehen, muss man dem Compiler mitteilen, dass man sich im Klaren ist, was man hier tut. Analog zum potentiellen Werteverlust bei der Umwandlung von elementaren Typen.

```
GeometrischesObjekt go = new Kreis(); // implizite Typumwandlung

if (go instanceof Kreis)
{
    // explizite Typumwandlung
    Kreis kreis = (Kreis) go;
    ...
}
```



cast-Operator

- Ein Objekt verändert nie seinen Typ und es "mutiert" niemals zu einem anderen Objekt einer anderen Klasse.
- Nur Referenzen und Referenzvariablen, d.h. die "Sicht" auf das Objekt, können sich vom Typ her ändern.
- Ein Auto bleibt ein Auto bleibt ein Auto.
- Ein Kreis bleibt ein Kreis bleibt ein Kreis.

```
GeometrischesObjekt go = new Kreis();  
Rechteck r = (Rechteck) go; // der Compiler akzeptiert das
```

Der Compiler akzeptiert, da wir durch den Cast-Operator ausdrücken: wir wissen, was wir tun...

- Das Laufzeitsystem reagiert nüchterner:

```
Exception in thread "main" java.lang.ClassCastException:  
Kreis cannot be cast to Rechteck  
at KreisTest.main(KreisTest.java:...)
```

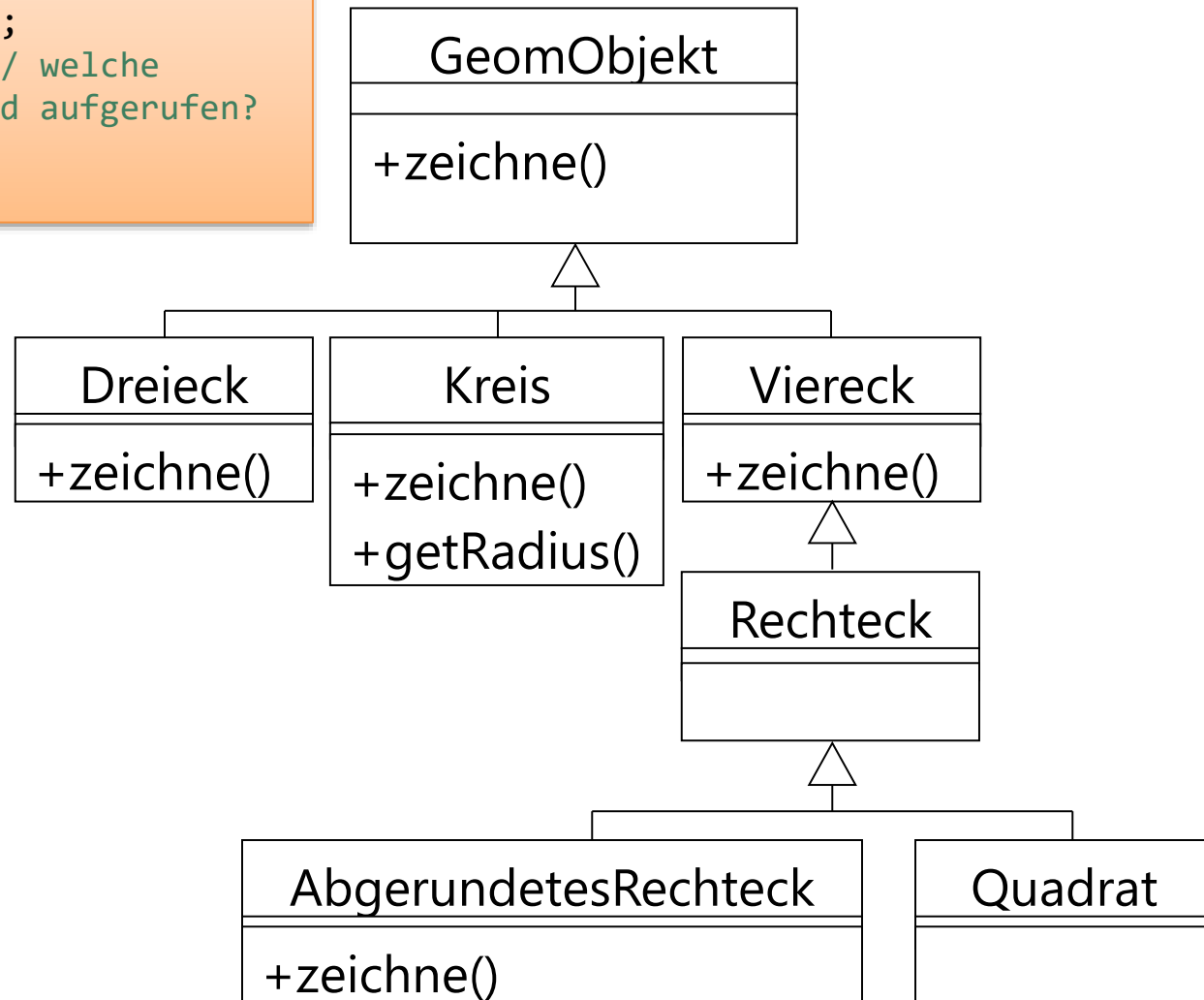
- Polymorph ist ein griech. Kunstwort.
- Poly – viel
- Morph – Gestalt
- Also vielgestaltig oder verschiedengestaltig.
- Man unterscheidet gerne zwischen universeller Polymorphie (aka dynamische Polymorphie) und ad-hoc Polymorphie (aka statische Polymorphie).
- Von ad-hoc Polymorphie spricht man, wenn die Anzahl der verschiedengestaltigen Möglichkeiten feststeht, z.B. beim Compilierungsvorgang.
- Hierunter fällt das Überladen (overloading) von Methoden.
- Universelle Polymorphie liegt dann vor, wenn die Anzahl der Möglichkeiten nicht beschränkt ist.
- Hierunter fällt das Überschreiben (overriding) von Methoden in einer uns unbekannten Anzahl von Unterklassen.

- Der Umgang des Compilers bzw. des Laufzeitsystems mit polymorphen (überschriebenen) Methoden.
- Dynamische Bindung liegt vor, wenn ein Methodenaufruf zur Laufzeit mit der für den vorliegenden Typ "besten" Implementierung aufgelöst wird.

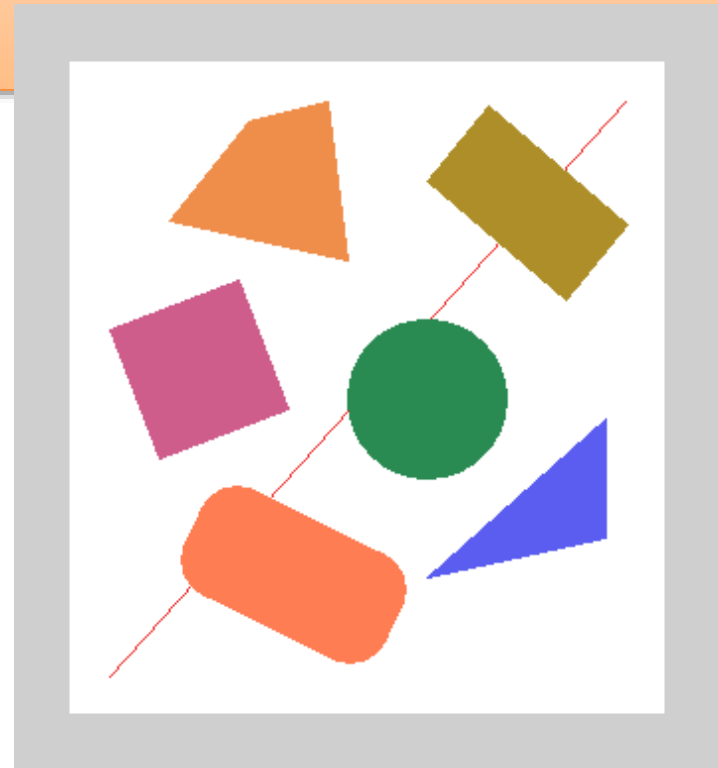
```
GeometrischesObjekt go = new Quadrat();  
go.zeichne(); // welche Methode wird aufgerufen?
```

- Die "beste" bezeichnet hier die technisch geeignetste.

```
GeometrischesObjekt go =  
    new Quadrat();  
go.zeichne(); // welche  
// Methode wird aufgerufen?
```



```
private static void zeichneBild(GeometrischesObjekt[] array)
{
    for (GeometrischesObjekt geometrischesObjekt : array)
    {
        geometrischesObjekt.zeichne();
    }
}
```



Die drei wichtigsten Begriffe für die Mechanismen, die Compilation und Laufzeit abgeleiteter, vererbter Klassen beeinflussen:

- Polymorphismus – je nach Kontext kann sich eine gleich aussehende Methode anders verhalten.
- Typkompatibilität – man kann in Referenzen von allgemeineren Typen verschiedene, zu diesem Typ kompatible Objekte ablegen.
- Dynamische Bindung – zur Laufzeit wird die "beste" Implementierung einer überschriebenen Methode gefunden.
- Frage: wie wirkt sich der Modifizierer `final` an einer Methode auf die dynamische Bindung aus?



11. Die Klasse Object

ARINKO[®]

- Jede Klasse ist direkte oder indirekte Unterklasse der Klasse Object.
- Natürlich mit Ausnahme der Klasse Object selbst, diese hat keine weiteren Oberklassen.
- Somit ist Object die Wurzel der Ableitungshierarchie.
- Wenn in der Klassendeklaration keine Ableitung definiert wurde, ergänzt der Compiler implizit `extends Object`.
- Somit ist garantiert, dass jede Klasse direkt oder indirekt von der Klasse Object erbt.
- In Object sind einige (zentrale) Eigenschaften hinterlegt, die jedes Java-Objekt können soll.
- Damit ist generell auch folgendes möglich:

```
Object o = beliebigeVariable; // alles ist kompatibel
```

<code>boolean equals(Object o)</code>	soll ein gegebenes und das aktuelle Objekt auf inhaltliche Gleichheit überprüfen
<code>int hashCode()</code>	soll einen möglichst eindeutigen Hashcode für das Objekt produzieren
<code>String toString()</code>	liefert eine Stringrepräsentation des aktuellen Objekts
<code>void finalize()</code>	kann vor der Destruktion vom Garbage Collector aufgerufen werden
<code>Object clone()</code>	soll unterstützen, eine Kopie des aktuellen Objektes anzufertigen
<code>wait() / notify()</code>	Methoden im Zusammenhang mit Threading
<code>getClass()</code>	Methode, die Meta-Information über die zugrunde liegende Klasse liefert

- Einige dieser Methoden sollte man in eigenen Klassen überschreiben, da das Default-Verhalten ungünstig ist.

Methode	Standardverhalten
<code>boolean equals(Object o)</code>	<code>return this == o;</code>
<code>int hashCode()</code>	liefert die Speicheradresse (nativ implementiert)
<code>String toString()</code>	Klassenname@hashCode
<code>void finalize()</code>	leer
<code>Object clone()</code>	wirft Exception
<code>wait() / notify()</code>	sind <code>final</code> und können nicht überschrieben werden
<code>getClass()</code>	ist <code>final</code> und kann nicht überschrieben werden

- `equals/hashCode/toString` sind Kandidaten für's Überschreiben.
- `finalize` sollte normalerweise nicht angerührt werden.
- `clone` bei Bedarf (selten)

- Der Gleichheitsoperator == überprüft den linken und rechten Operand auf Wertgleichheit, nicht Inhaltsgleichheit eines Objektes.

```
Kreis k1 = new Kreis();  
Kreis k2 = new Kreis();  
// k1 und k2 haben denselben Objektzustand (defaults)  
// sind aber verschiedene Objekte  
if (k1 == k2) // false  
{...}
```

- Um auf inhaltsgleiche Objekte zu reagieren, benützt man die equals()-Methode.

```
Kreis k1 = new Kreis();  
Kreis k2 = new Kreis();  
// k1 und k2 haben denselben Objektzustand (defaults)  
// sind aber verschiedene Objekte  
if (k1.equals(k2))  
{}
```

- Das Problem ist: was bedeutet eigentlich "inhaltsgleich"?
- Das ist ein fachliches Thema und kann somit nicht automatisiert von Java beantwortet werden.
- Zwei Kreise könnten dann gleich sein:

Sie haben den selben
Mittelpunkt und Radius.

Sie haben den selben
Mittelpunkt und Radius
und die gleichen Farben

Sie haben nur die
gleichen Farben
(unwahrscheinlich)

Sie haben nur den
gleichen Radius.

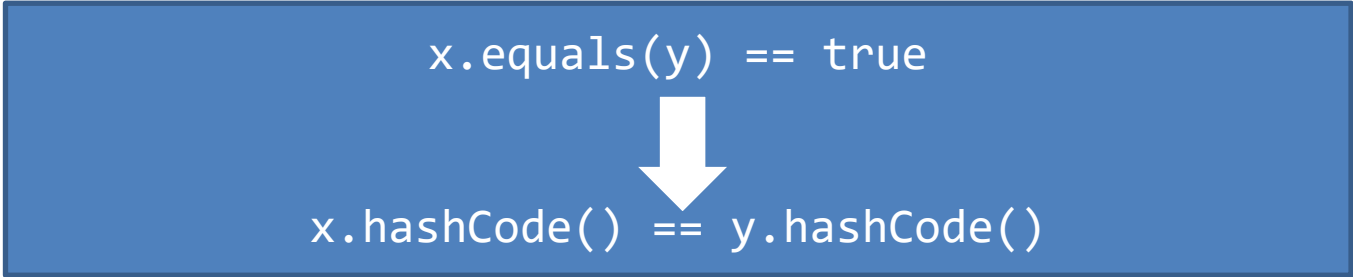
etc.

```
public boolean equals(Object obj)
{
    // nicht vollumfänglich korrekt, aber nun gut...
    // wir gehen nur auf Mittelpunkt und Radius
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (obj instanceof Kreis)
    {
        Kreis that = (Kreis) obj;
        if (!this.mittelpunkt.equals(that.mittelpunkt))
            return false;
        if (this.radius != that.radius)
            return false;
        return true;
    }
    return false;
}
```

Für die Implementierung von equals gelten folgende Vorgaben, die in der Spezifikation der equals-Methode in der Klasse Object hinterlegt sind:

- reflexiv:
`x.equals(x) == true`
- symmetrisch:
`x.equals(y) == y.equals(x)`
- transitiv:
`x.equals(y) & y.equals(z) == x.equals(z)`
- konsistent:
mehrfacher Aufruf von `x.equals(y)` sollte immer das selbe Resultat ergeben
- `x.equals(null) == false`

- hashCode und equals werden oft zusammen benutzt (z.B. bei der Einsortierung von Objekten in Collections).
- Wir kommen darauf nochmals zurück.
- Man merke sich aber bereits folgende Implikation:



```
x.equals(y) == true
      ↓
x.hashCode() == y.hashCode()
```

- Lesart: wenn zwei Objekte inhaltsgleich sind, sollten sie auch den selben Hashcode haben.
- Es folgt bewusst NICHT: wenn zwei Objekte den selben Hashcode haben, so sind sie auch inhaltsgleich.

