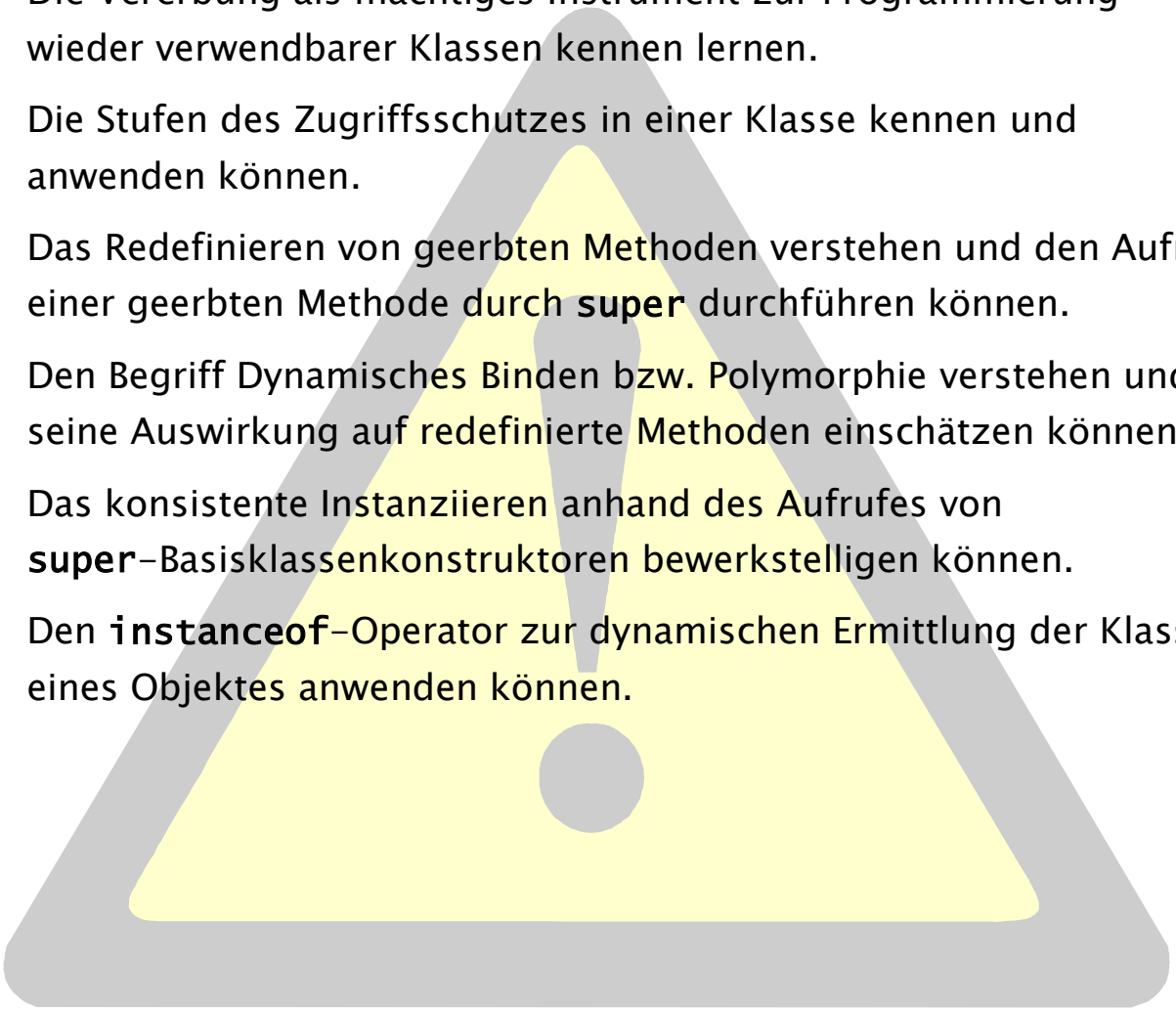
Informatik: Vererbung, Zugriffsschutz, dynamisches Binden

Ziele

- Die Vererbung als mächtiges Instrument zur Programmierung wieder verwendbarer Klassen kennen lernen.
- Die Stufen des Zugriffsschutzes in einer Klasse kennen und anwenden können.
- Das Redefinieren von geerbten Methoden verstehen und den Aufruf einer geerbten Methode durch `super` durchführen können.
- Den Begriff Dynamisches Binden bzw. Polymorphie verstehen und seine Auswirkung auf redefinierte Methoden einschätzen können.
- Das konsistente Instanziieren anhand des Aufrufes von `super`-Basisklassenkonstruktoren bewerkstelligen können.
- Den `instanceof`-Operator zur dynamischen Ermittlung der Klasse eines Objektes anwenden können.



Ausgangspunkt

Klassendeklaration

```
public class OffenerZaehler
  extends Object
{
  protected int zaehlerstand = 0;

  public void inkrementiere() {
    this.zaehlerstand++;
  }

  public int getZaehlerstand() {
    return this.zaehlerstand;
  }

  public void setNull() {
    this.zaehlerstand = 0;
  }
}
```

Verwendung

```
OffenerZaehler z =
  new OffenerZaehler();

for (int i = 0; i < 10; i++) {
  z.inkrementiere();
  System.out.println(
    z.getZaehlerstand());
}
```

Ergibt: 1 2 3 4 5 6 7 8 9 10

- Wenn kein **extends** bei der Klassendeklaration vorhanden ist, wird die Klasse von der Klasse **Object** abgeleitet.
- Mit **proteced** gekennzeichnete Variablen und Methoden sind in abgeleiteten Klassen sichtbar, in angelegten Objekten aber nicht (siehe hinten).

1. Erweiterungswunsch: MerkeZaehler

Klasse die wie OffenerZaehler funktioniert aber noch zusätzlich sich den Zählerstand merken (merke) und auf diesen zurückspringen (zurueck) kann.

Klassendeklaration

```
public class MerkeZaehler
  extends OffenerZaehler
{
  protected int merker = 0;

  public void merke() {
    this.merker = this.zaehlerstand;
  }

  public void zurueck() {
    this.zaehlerstand = this.merker;
  }
}
```

Verwendung

```
MerkeZaehler z = new
  MerkeZaehler();

z.inkrementiere();
z.merke();
z.inkrementiere();
z.inkrementiere();
z.zurueck();
z.inkrementiere();
z.setNull();
```

- Durch Ableitung wird der Quelltext einer erweiterten Klasse auf die Unterschiede zur grundlegenden Klasse reduziert.
- OffenerZaehler ist *Basisklasse* (engl. *super class*) von MerkeZaehler.
- MerkeZaehler ist *abgeleitete Klasse* (engl. *derived class*) von OffenerZaehler.
- Bei einem Methodenaufruf wird zuerst in der Klasse des Objektes nach Methode gesucht. Sollte dort keine Methode gefunden werden, wird in der Basisklasse, Basisklasse der Basisklasse, usw. gesucht.

	OffenerZaehler	MerkeZaehler
Konstruktor	automatisch	automatisch
Datenelemente	`int zaehlerstand`	← erbt `int merker`

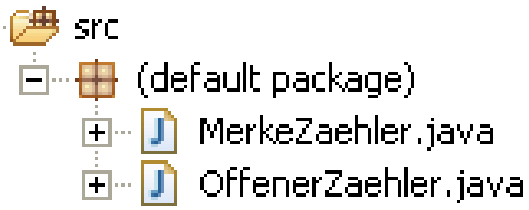
Methoden	`void inkrementiere()`
	`void setNull()`
	`int getZaehlerstand()`
		← erbt
		← erbt
		← erbt
		`void merke()`
		`void zurueck()`

Stufen des Zugriffsschutzes

`private`	Zugriffsmöglichkeit nur innerhalb der Klasse.
package	(ohne Angabe) wie **`private`**, nur für Klassen desselben Package wie **`public`**.
`protected`	wie *package* und zusätzlich für alle direkt oder indirekt abgeleiteten Klassen wie **`public`**. Für angelegte Objekte außerhalb des Package wie **`private`**.
`public`	Uneingeschränkter Zugriff.

`z.merker = 3;` würde funktionieren, wenn Zuweisung in einer Klasse desselben Package erfolgt.

`z.merker = 3;` würde nicht funktionieren, wenn Zuweisung in einer Klasse außerhalb des Package erfolgt.



2. Erweiterungswunsch: AnschlagZaehler

Klasse die wie OffenerZaehler funktioniert aber nur bis zu einem bestimmten Anschlag hochzählt und dort stehen bleibt. Im Zähler soll der Anschlag (anschlag) gespeichert werden. Der Anschlag wird durch den Konstruktor initialisiert.

Klassendeklaration

```
public class AnschlagZaehler
  extends OffenerZaehler
{
  protected int anschlag = 0;

  public AnschlagZaehler(int anschlag) {
    this.anschlag = anschlag;
  }

  public int getAnschlag() {
    return this.anschlag;
  }

  public void inkrementiere() {
    if (this.zaehlerstand < this.anschlag)
      this.zaehlerstand++;
  }

}
```

Verwendung

```
AnschlagZaehler z = new
  AnschlagZaehler(3);

z.inkrementiere();
z.inkrementiere();
z.inkrementiere();
z.inkrementiere();
z.setNull();
```

- Klasse redefiniert die Methode inkrementiere().
- Name und Parameterliste müssen exakt übernommen werden.
- Zugriffsschutz darf gelockert werden.
- In einer abgeleiteten Klasse können neue Datenelemente und Methoden definiert bzw. redefiniert, es kann aber keine geerbte Methode gelöscht werden.

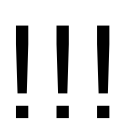
	OffenerZaehler	AnschlagZaehler
Konstruktor	automatisch	AnschlagZaehler(**int**)
Datenelemente	**int** zaehlerstand	← erbt **int** anschlag
Methoden	**void** inkrementiere() **void** setNull() **int** getZaehlerstand()	**void** inkrementiere() ← erbt ← erbt **int** getAnschlag()

Dynamisches Binden redefinierter Methoden (Polymorphie)

```
OffenerZaehler z = new AnschlagZaehler(3);
z.inkrementiere();
```

- Eine Objektvariable mit dem Typ der Basisklasse kann alle abgeleiteten Objekte aufnehmen.
- Zur Laufzeit (dynamisch) wird die richtige Methode passend zum tatsächlich angelegten Objekt gesucht.

```
Object o = new AnschlagZaehler(3);
o.inkrementiere();
```



- Die Klasse `Object` besitzt kein `inkrementiere()`.

3. Erweiterungswunsch: RücksprungZaehler

Klassendeklaration

```
public class RuecksprungZaehler
  extends AnschlagZaehler
{
  public RuecksprungZaehler(int anschlag) {
    super(anschlag);
  }
  public void inkrementiere() {
    if (this.zaehlerstand == this.anschlag)
      this.zaehlerstand = 0;
    else
      this.zaehlerstand++;
  }
}
```

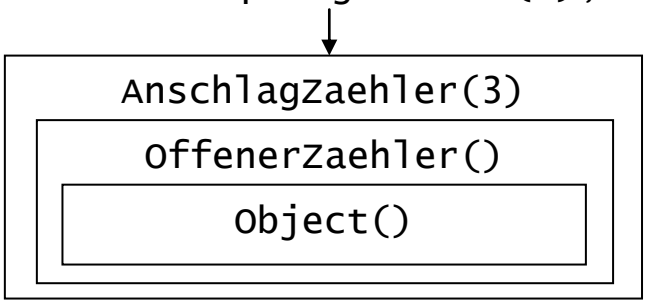
Verwendung

```
RuecksprungZaehler z = new
  RuecksprungZaehler(3);
z.inkrementiere();
z.inkrementiere();
z.inkrementiere();
z.inkrementiere();
z.inkrementiere();
```

- `super(anschlag)` ruft den Basisklassenkonstruktor auf und sorgt dafür, dass das Basisklassenobjekt sich ordnungsgemäß initialisiert.
- Muss als erste Anweisung im Konstruktorrumpf stehen.
- Wenn kein Basisklassenkonstruktoraufruf eingefügt wurde, fügt Compiler automatisch Aufruf des parameterlosen Defaultkonstruktors `super()` ein.

Aufruf von folgenden Konstruktoren:

```
RuecksprungZaehler z = new RuecksprungZaehler(3);
```



OffenerZaehler	AnschlagZaehler	RuecksprungZaehler
automatisch	AnschlagZaehler(**int**)	RuecksprungZaehler(**int**)
int zaehlerstand	← erbt **int** anschlag	← erbt ← erbt
void inkrementiere() **void** setNull() **int** getZaehlerst()	**void** inkrementiere() ← erbt ← erbt **int** getAnschlag()	**void** inkrementiere() ← erbt ← erbt ← erbt

4. Erweiterungswunsch: Bezug auf die Basisklasse

super kann in normalen Methoden eingesetzt werden, um das jeweilige Basisklassenobjekt anzusprechen:

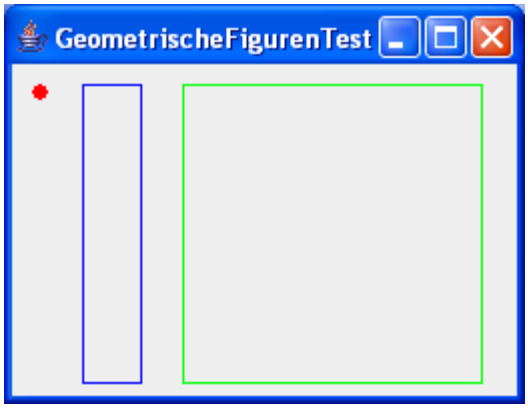
```
public class RuecksprungZaehler extends AnschlagZaehler
{
  …
  public void inkrementiere() {
    if (super.getZaehlerstand() == super.getAnschlag())
      super.setNull();
    else
      super.inkrementiere();
  }

  …
}
```

super in obiger Methode teils überflüssig:

```
  public void inkrementiere() {
    if (getZaehlerstand() == getAnschlag())
      setNull();
    else
      super.inkrementiere();
  }
```

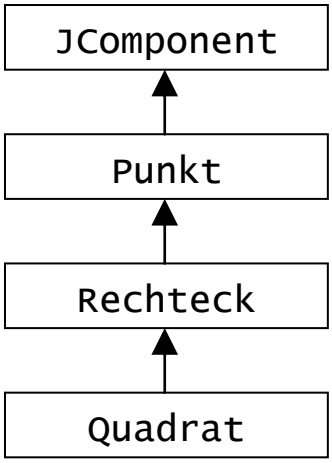
Beispiel: Geometrische Figuren Punkt, Rechteck, Quadrat

Basisklasse `javax.swing.JComponent`



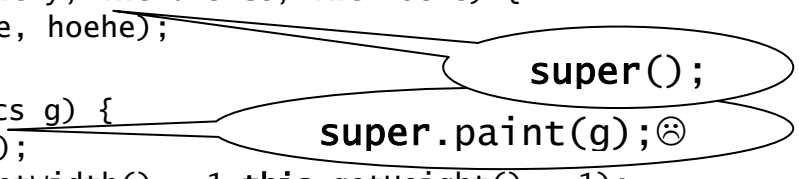
- `JComponent()`
  Default-Konstruktor
- `void paint(Graphics g)`,
  `void repaint()`
  übernehmen das Zeichnen der Figur
- `int getX()`, `int getY()`,
  `int getWidth()`, `int getHeight()`
- `void setBounds(int x, int y, int width, int height)`

```
import javax.swing.*;                    // JComponent
import java.awt.*;                       // Color, Graphics
public class Punkt extends JComponent
{
   private static final int PUNKT_GROESSE = 8;
   protected Color farbe = Color.BLACK;
   // Defaultkonstruktor
   public Punkt() {
   }
   public Punkt(int x, int y) {
      setBounds(x, y, PUNKT_GROESSE, PUNKT_GROESSE);
   }
   public void setFarbe(Color farbe) {
      this.farbe = farbe;
      repaint();
   }
   public Color getFarbe() {
      return this.farbe;
   }
   public void paint(Graphics g) {
      g.setColor(this.farbe);
      g.fillOval(0,0,PUNKT_GROESSE,PUNKT_GROESSE);
   }
}
```



Klasse Rechteck

```
import java.awt.*;              // Graphics
public class Rechteck extends Punkt
{
   public Rechteck(int x, int y, int breite, int hoehe) {
      setBounds(x, y, breite, hoehe);
   }
   public void paint(Graphics g) {
      g.setColor(this.farbe);
      g.drawRect(0,0,this.getWidth() - 1,this.getHeight() - 1);
   }
}
```



Klasse Quadrat

```
public class Quadrat extends Rechteck
{
   public Quadrat(int x, int y, int breite) {
      super(x, y, breite, breite);
   }
   public void setBounds(int x, int y, int breite) {
      super.setBounds(x, y, breite, breite);
   }
   public void setBounds(int x, int y, int breite, int hoehe) {
      super.setBounds(x, y, breite, breite);
   }
}
```

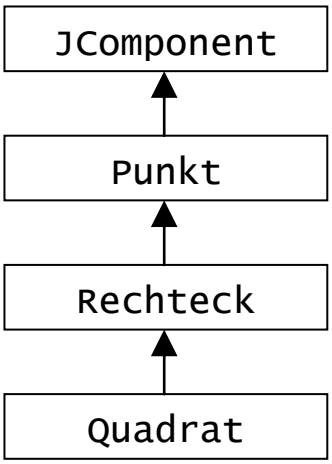
Hauptprogramm (Konstruktor)

```
public GeometrischeFigurenTest() {
   super("GeometrischeFigurenTest");
   setBounds(0,0,260,200);
   Container contentPane = this.getContentPane();
   contentPane.setLayout(null);
   Punkt p = new Punkt(10,10);
   p.setFarbe(Color.RED);
   p.setBounds(10,10,100,100);
   contentPane.add(p);
   Rechteck r = new Rechteck(35,10,30,150);
   r.setFarbe(Color.BLUE);
   contentPane.add(r);
   Quadrat q = new Quadrat(85,10,150);
   q.setFarbe(Color.GREEN);
   contentPane.add(q);
   setVisible(true);
   …
}
```

Zusammenfassung

	JComponent	Punkt	Rechteck	Quadrat
Konstr.	`JComponent()`	`Punkt()` `Punkt(int, int)`	`Rechteck(int, int, int, int)`	`Quadrat(int, int, int)`
Daten		`Color farbe`	← erbt	← erbt
Meth.	`void paint(Graphics)`	`void paint(Graphics)`	`void paint(Graphics)`	← erbt
	`void setBounds(int, int, int, int)`	← erbt	← erbt	`void setBounds(int, int, int, int)`
		`void setFarbe(int)`	← erbt	← erbt
		`Color getFarbe()`	← erbt	← erbt
				`void setBounds(int, int, int)`

Dynamische Typinformation durch instanceof

```
Punkt p = new Rechteck(35,10,30,150);
p instanceof JComponent    ergibt true
p instanceof Punkt         ergibt true
p instanceof Rechteck      ergibt true
p instanceof Quadrat       ergibt false
```

Liefert **true**, falls Objekt zum angegebenen Typ kompatibel ist.