Informatik II

Vorlesung am D-BAUG der ETH Zürich

Vorlesung 8, 25.4.2016

Objektorientierte Programmierung. Motivation Zeichenprogramm, Vererbung, Polymorphie,
Fallstudie Numerische Integration: Rechteck-, Trapez- und Simpson-Regel

Letzte Vorlesung / Übung

Interne Repräsentation als verkettete Liste

PointInPolygon Algorithmus

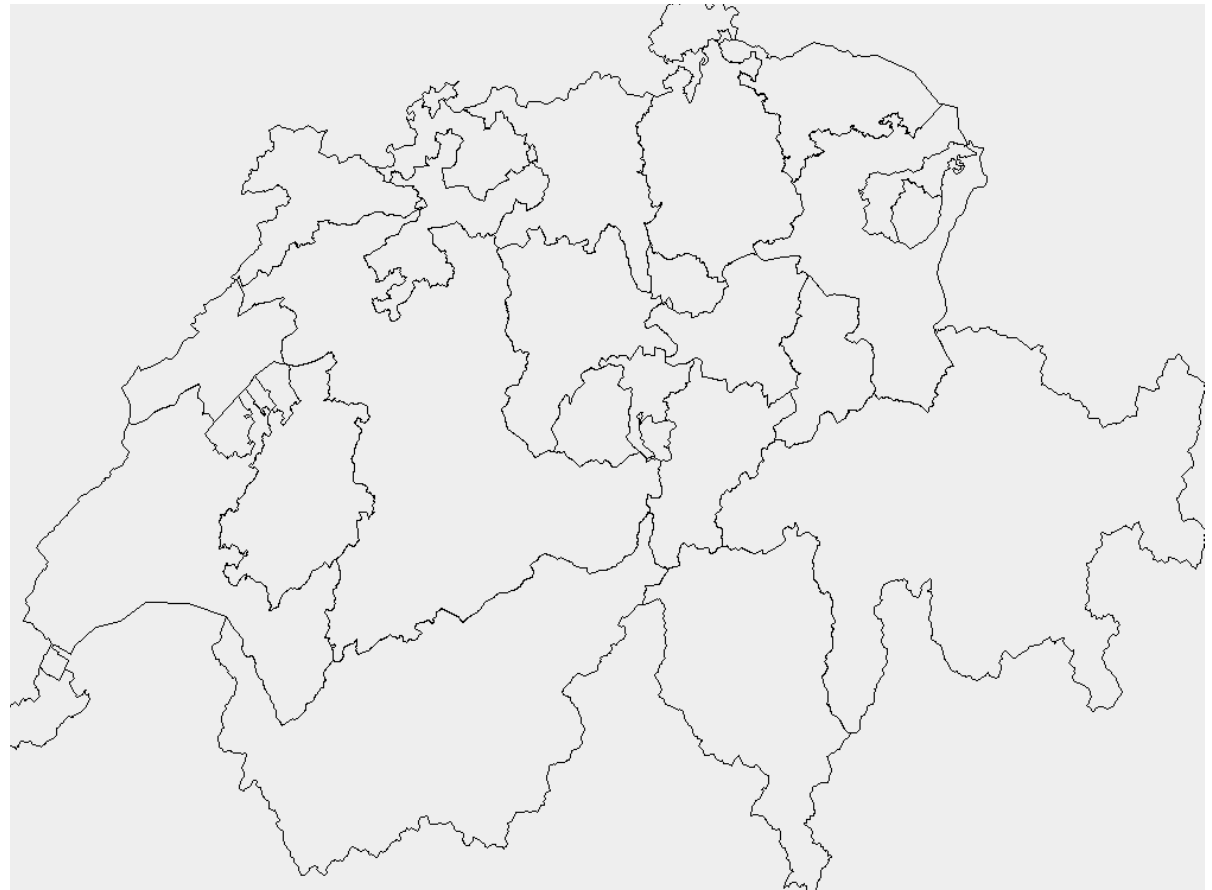
```
public class Polygon {
   public boolean PointInPolygon (int px, int py) {...}
   public void DrawPolygon(ImageViewer p) {...}
}
```

Diskretisierung → Linien zeichnen

Listen von Polygonen

```
ArrayList<Polygon> polys = new ArrayList<Polygon>();
Polygon poly = new Polygon(name);
...
polys.add(poly);
....
for (Polygon poly: polys){
   poly.DrawPolygon(v);
}
```

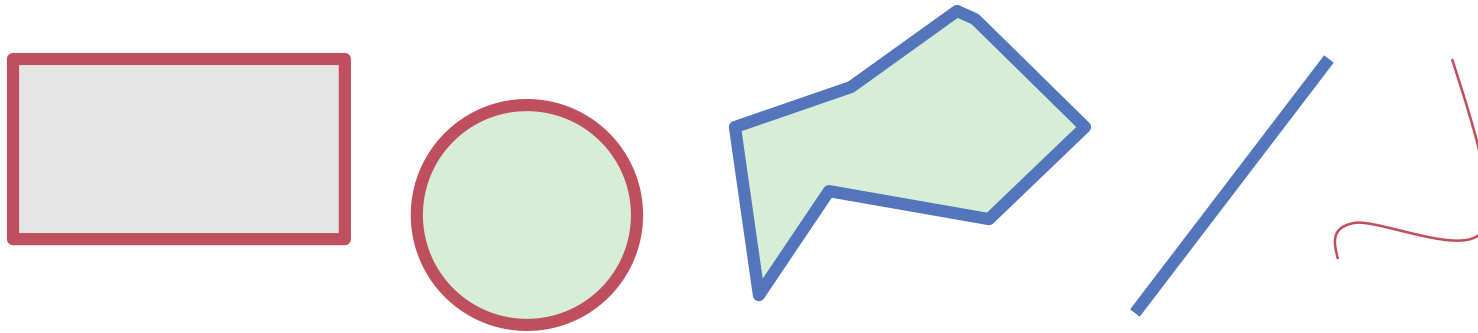
poly Zürich 8.669431372108662,47.67467431064423,0 ....
poly Bern 7.558341989699091,47.32236641629984,0 ...
....



Vererbung – Motivation

Beispielhaftes Ziel: Grafikbibliothek mit erweitertem Fundus an geometrischen Figuren

- Rechteck, Kreis, Polygon, Linie, Kurve etc.



Gemeinsame Eigenschaften

- Randfarbe, Randdicke, Füllfarbe, Muster, Offset, etc.

Gemeinsame Operationen

- Zeichnen, Füllen, PointIn, Schneiden mit Geraden, etc.

Vererbung – Motivation

Unterschiede der Figuren:

Repräsentation

Rechteck: x,y,w,h,

Kreis: x,y,r,

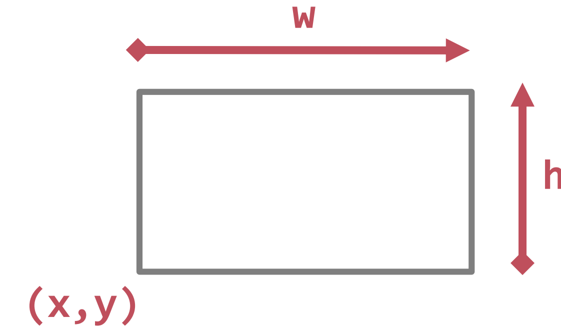
Polygon: Liste von Eckpunkten

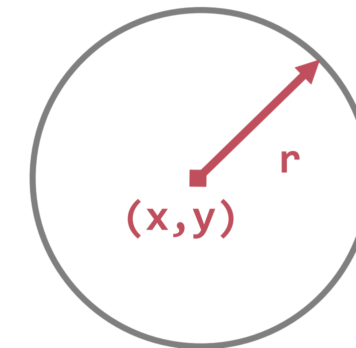
Implementation der Operationen

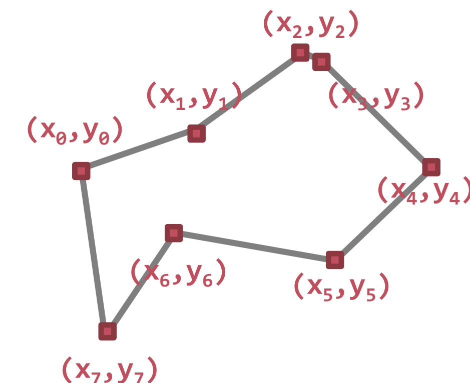
Zeichnen / Füllen verschieden aufwändig

Schnitt mit Gerade oder Erkennung eines innenliegenden Punktes anders zu implementieren

etc.







Vererbung – Motivation

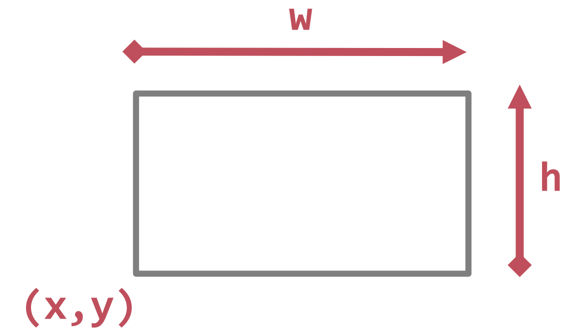
Annahme: wir wollen ein Zeichenprogramm schreiben, welches z.B. über eine (verkettete) Liste auf seine geometrischen Objekte zugreift.

Wie lässt sich das realisieren?

Wir versuchen es (live coding).

Rechteck

```
public class Rectangle {
   Color color;
   int x,y,w,h;

   public Rectangle(Color c, int X, int Y, int W, int H){
      x = X; y = Y; w = W; h = H;
      color = c;
   }

   public void Draw(ImageViewer v) {
      v.setColor(color);
      v.line(x,y,x,y+h-1);
      v.line(x,y,x+w-1,y);
      v.line(x,y+h-1,x+w-1,y+h-1);
      v.line(x+w-1,y,x+w-1,y+h-1);
   }
}
```

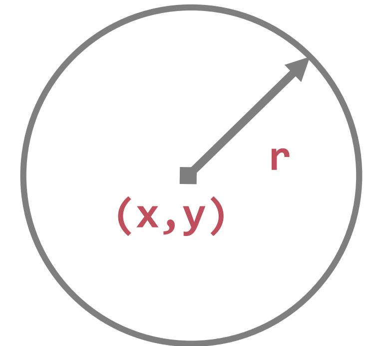


Kreis

```
public class Circle {
   Color color;
   int x, y, r;

   public Circle(Color c, int X, int Y, int R){
      x = X; y = Y; r = R;
      color = c;
   }

   public void Draw(ImageViewer v){
      v.setColor(color);
      v.ellipse(x,y,r,r);
   }
}
```

Gemeinsame Eigenschaft von Kreis und Rechteck

Unterschiedliche Repräsentation von Kreis und Rechteck



Gemeinsame angebotene Funktionalität

Interne Repräsentation und Implementation unterschiedlich für Rechteck und Kreis.

Verwalten und zeichnen...

```
ArrayList<Circle> circles = new ArrayList<Circle>();
ArrayList<Rectangle> rectangles = new ArrayList<Rectangle>();
...
Circle c = new Circle(Color.yellow, x,y,r);
circles.add(c);
...
Rectangle r = new Rectangle(Color.red, x,y,w,h);
rectangles.add(r);
...
for (Circle c: circles){
   c.Draw(v);
}
for (Rectangle r: rectangles){
   r.Draw(v);
}
```

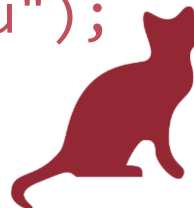
Alles ist repliziert
→ Redundanz

Ich will aber auch noch Polygone !
Und nun?

Einfache Tatsachen

```
class Hund {

  public void Hello(){
    System.out.println("Wau");
  }
}
```

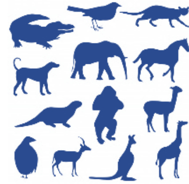
```
class Katze{

  public void Hello(){
    System.out.println("Miau");
  }
}
```



```
Hund h = new Hund();
Katze k = new Katze();

h.Hello();  // "Wau"
k.Hello();  // "Miau"

h = k; // nicht erlaubt !!!
```

Magische Beobachtungen

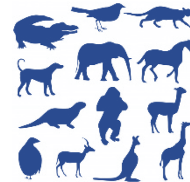
```
class Tier{
   public void Hello(){
   }
}
```



```
class Hund extends Tier
{
   public void Hello(){
      System.out.println("Wau");
   }
}
```

```
class Katze extends Tier
{
   public void Hello(){
      System.out.println("Miau");
   }
}
```

```
Hund h = new Hund();
Katze k = new Katze();

h.Hello();  // "Wau"
k.Hello();  // "Miau"

Tier t;
```



```
t = k;
k.Hello(); // "Miau"
```

```
t = h;
t.Hello(); // "Wau"
```

Magische Beobachtungen II

```
class Tier{
public void Hello(){
   }
}

class Hund extends Tier{
   public void Hello(){
      System.out.println("Wau");
   }
}

class Katze extends Tier {
   public void Hello(){
      System.out.println("Miau");
   }
}
```

```
public static void Chat(Tier t1, Tier t2)
{
   t1.Hello();
   System.out.String(" , ");
   t2.Hello();
}

public static Tier Any(){
   double r = Math.random()*2;
   if (r < 1) return new Hund();
   else return new Katze();
}

Hund h = new Hund();
Katze k = new Katze();

Chat(h,k); // "Wau , Miau";
Chat(Any(), Any());
```

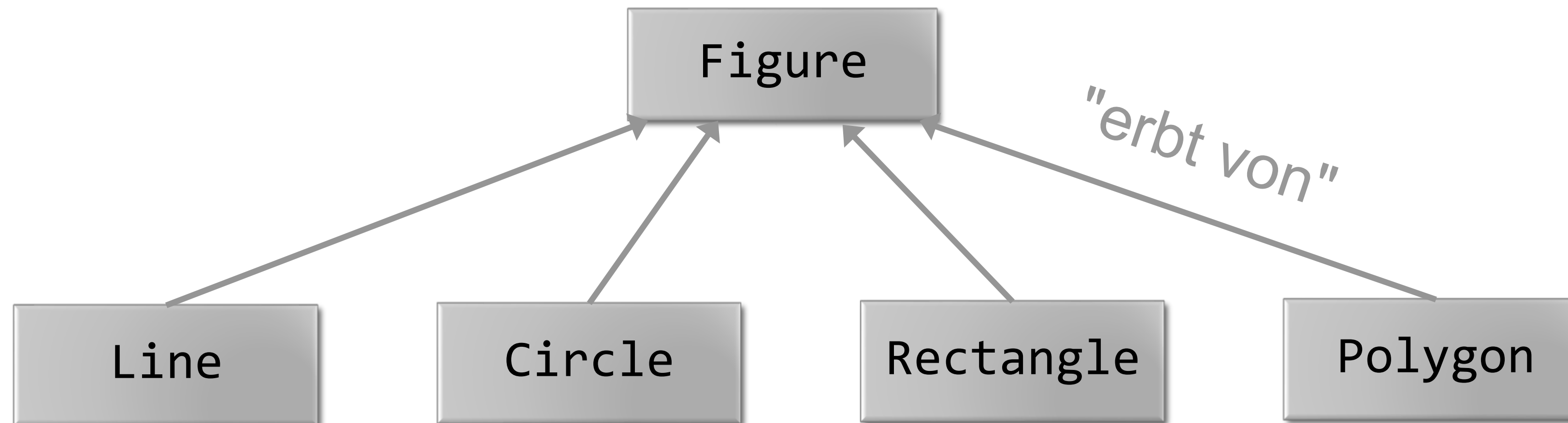
Gemeinsame Eigenschaften

```
class Tier{
   int gewicht;
   public void Hello(){
   }
}

class Hund extends Tier{
   public void Hello(){
      System.out.println("Wau");
   }
}

class Katze extends Tier {
   public void Hello(){
      if (gewicht < 10)
         System.out.println("Miau");
      else
         System.out.println("Grrrr");
   }
}
```

Vererbung

Zurück zu den Figuren

Darstellung jeder geometrischen Figur als **Erweiterung** eines zugrunde liegenden **Grundtyps** `Figure`



Gemeinsame Eigenschaften verbleiben (nur) in der Basisklasse `Figure`.

Klassen können Eigenschaften *(ver)erben*

```
public class Figure {
    Color col = Color.black;


    public void setColor(Color C){
        col = C;
    }
    public void Draw(){
    }
}
```

Datenfelder, welche für alle Figuren deklariert sind

Methoden, welche für alle Figuren deklariert sind

Klassen können Eigenschaften *(ver)erben*

```
public class Circle extends Figure {
  int x,y,r;

  // Konstruktor eines Kreises
  public Circle(Color c, int X, int Y, int R) {
    setColor(c);
    x = X; y = Y; r = R;
  }

  ...
}
```

Datenfelder, welche nur für `Circle` deklariert sind

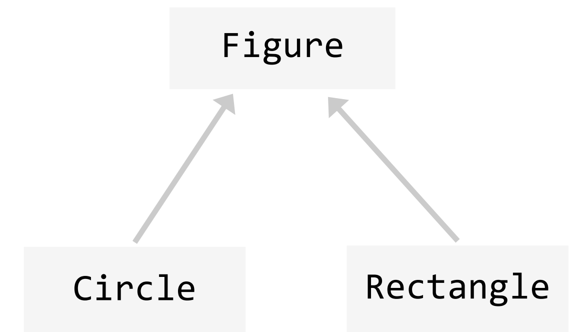
Klassen können Eigenschaften *(ver)erben*

```java
public class Rectangle extends Figure {
   int x,y,w,h;

   // Konstruktor des Rechtecks
   Rectangle(Color c, int X, int Y, int W, int H) {
      setColor(c);
      x = X; y = Y; w = W; h = H;
   }
   ...
}
```

Datenfelder, welche nur für `Rectangle` deklariert sind

Klassen können Eigenschaften *(ver)erben*

```
public class Figure { ... }
public class Circle extends Figure { ... }
public class Rectangle extends Figure { ... }

...
Rectangle rectangle=new Rectangle(100,100,200,100);
Circle circle= new Circle(100,100,50);
circle.SetColor(Color.blue);
rectangle.SetColor(Color.red);
...
```



Aufruf der Methoden der Basisklasse `Figure`

➔ Wiederverwendbarkeit gemeinsam nutzbaren Codes durch ***Generalisierung***

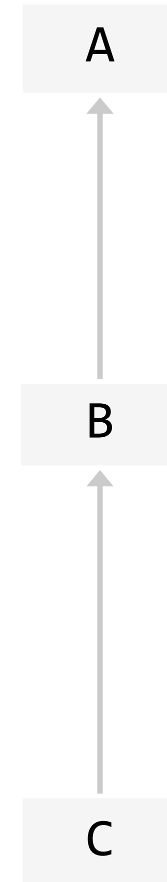
Vererbung – Nomenklatur

```
class A {
      ...
}

class B extends A {
      ...
}

class C extends B {
      ...
}
```

Basisklasse (Superklasse)

Abgeleitete Klasse (Subklasse)

«B und C *erben von* A»
«C *erbt von* B»



Vererbung – Kompatibilität

Ein abgeleitetes Objekt kann überall dort verwendet werden, wo ein Basisobjekt gefordert ist, aber nicht umgekehrt.

```
Rectangle rectangle = new Rectangle(0,0,10,10);
Figure figure = rectangle;        // ok: Rectangle extends Figure
rectangle.setColor(Color.red);   // ok: Rectangle verwendet eine Methode von seiner Basisklasse
Rectangle r = new Circle(0,0,10); // type mismatch: cannot convert from Circle to Rectangle
Figure c = new Circle(0,0,10);    // ok: Circle extends Figure
Circle circle = c;                // type mismatch: cannot convert from Figure to Circle
```

Vererbung – Sichtbarkeit

In der abgeleiteten Klasse können Variablen und Methoden der Basisklasse direkt verwendet werden

Voraussetzung:
Sichtbarkeit gewährleistet

Sichtbarkeiten nach Modifizierer

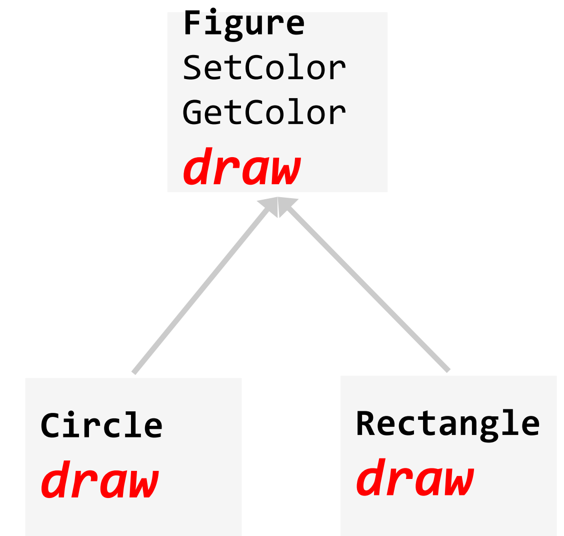
Modifizierer	Klasse	Paket	Sub-Klasse	Global
public	✓	✓	✓	✓
protected	✓	✓	✓	
(keiner)	✓	✓		
private	✓			

```
public class Rectangle extends Figure {
  ...
  // zusätzlicher Konstruktor
  Rectangle (Color col; int rx, int ry, int rw, rh) {
    x = rx; y = ry; w = rw; h = rh;
        SetColor(col);
  }
  ...
}
```

ruft die Methode SetColor in der Basisklasse `figure` auf

Polymorphie in Java

```
public class Figure {
   ...
   public void Draw(ImageViewer v) { }
}

public class Circle extends Figure {
   ...
   public void Draw(ImageViewer v) {...} // draw circle
}

public class Rectangle extends Figure {
   ...
   public void Draw(ImageViewer v) {  ... } // draw rect
}
```



Die Magie der Polymorphie

Polymorphie (in Java): Deklariert man eine (nicht statische) Methode mit gleichem Namen und gleicher Signatur in abgeleiteten Klassen *und* in der gemeinsamen Basisklasse, so wird *zur Laufzeit* automatisch über die auszuführende Variante entschieden.

Man sagt, die Methode wird in der abgeleiteten Klasse *überschrieben*.

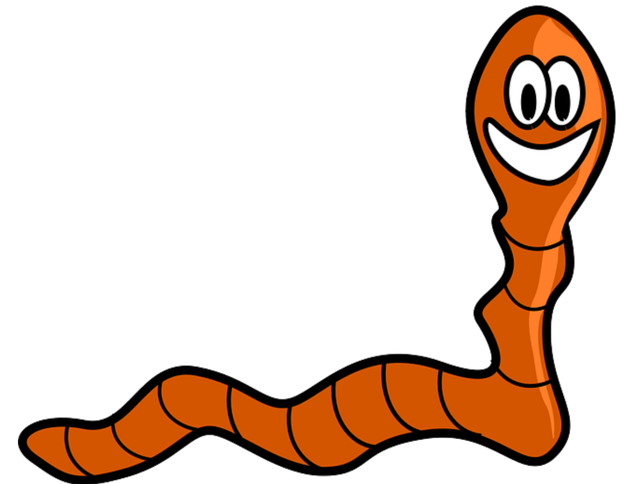
Polymorphie

Bei überschriebenen Methoden wird der *dynamische* Typ gewählt («dynamic dispatching»)

```
ImageViewer v = new ImageViewer(400,400);

Figure f1 = new Rectangle(100,100,200,100);
Figure f2 = new Circle(100,100,50);
f1.draw(v); // wählt automagisch draw von Rectangle
f2.draw(v); // wählt automagisch draw von Circle
```

Ziel erreicht!

```
ArrayList<Figure> figures = new ArrayList<Figure>();
...
figures.add(new Rectangle(...));
figures.add(new Circle(...));
...
for (Figure f : figures)
    f.Draw(v);
```



Fallstudie: Numerische Integration

Ziel: Erstellung eines Softwareframeworks zur numerischen Integration («Quadratur») einer gegebenen Funktion $f: \mathbb{R} \to \mathbb{R}$.

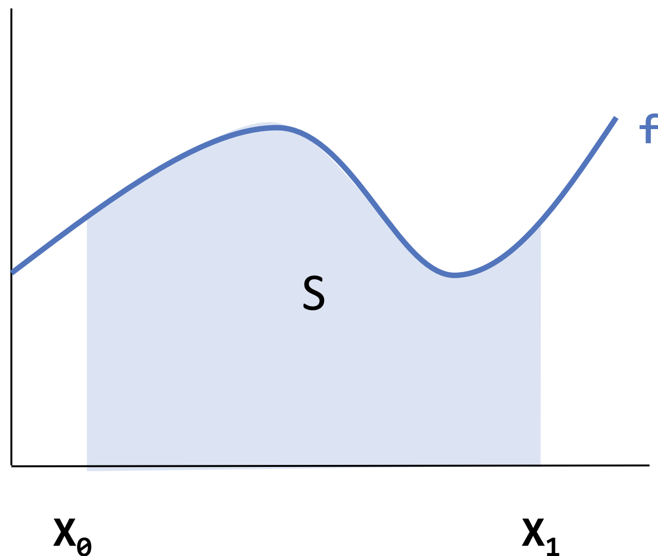
Problem aus Sicht der Softwareentwicklung: in Java gibt es keine Variablen von Funktionstyp. Wie können wir trotzdem ein generisches Tool bauen?

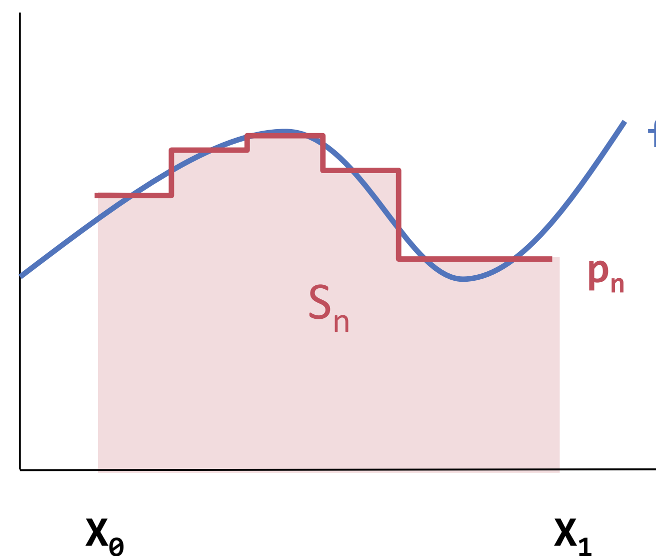
Antwort: wir verwenden Vererbung und Polymorphie für die generische Darstellung von reellwertigen Funktionen.

Numerische Integration

Einfachster Ansatz: Approximiere die Funktion f im gewünschten Integrationsintervall $[x_0, x_1]$ durch eine stückweise konstante Funktion p_n mit n Stücken.

Approximiere das Integral I von f durch das Integral I_n von p_n





Generische Funktionenklasse

```
public abstract class Function {
    public abstract double Evaluate(double x);
}
```

Abstrakte Klassen können nicht instanziert werden

Abstrakte Funktionen benötigen keinen Funktionsrumpf, müssen jedoch von erbenden Klassen, welche nicht abstract sind, implementiert werden.

Generischer Integrierer

```
public abstract class Integrator {
   int n;
   public void SetNumberPieces(int pieces) {
      n = pieces;
   }
   public abstract double Integrate(Function f, double x0, double x1);
}
```

Konkretisierung: Parabel

$$f(x) = x^2$$

```
class Square extends Function{

    public double Evaluate (double x) {
        return x*x;
    }
}
```

Konkretisierung: Dichte der Normalverteilung

```
class Normal extends Function{
   double mu;
   double sigma;

   Normal(double m, double s)  {
      mu = m; sigma = s;
   }

   public double Evaluate(double x) {
      return  1/Math.sqrt(2*Math.PI)/sigma
              * Math.exp(-(x-mu)*(x-mu)/(2*sigma*sigma));
   }
}
```

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Konkretisierung: Integrierer mit Rechteckregel

```
public class RectangleIntegrator extends Integrator {

    RectangleIntegrator(int pieces){
        n= pieces;
    }

    public double Integrate(Function f, double x0, double x1)   {
        double sum = 0;
        double width=(x1-x0)/n;                   // Intervallbreite
        for (int i = 0; i<n; ++i) {
            double x = x0 + (i+0.5)*width;        // Mitte des Intervalls
            double y = f.Evaluate(x);             // Abgreifen des Funktionswertes
            sum += y*width;                       // Rechteckinhalt addieren
        }
        return sum;
    }
}
```

Testbeispiel

```
Integrator integrator = new RectangleIntegrator(100);
Function sq = new Square();
Function N = new Normal(0,1);
System.out.println("I(sq,0,2)= " + integrator.Integrate(sq, 0, 2) );
for (int i=1; i<=3;++i){
        System.out.print("I(" +(-i) + "," + i + ")= ");
        System.out.println(integrator.Integrate(N,-i,i));
}
```

Ausgabe:

- `I(sq,0,2)= 2.6666000000000003`
- `I(N,-1,1)= 0.6826975580161088`
- `I(N,-2,2)= 0.9545141330225693`
- `I(N,-3,3)= 0.9973041900876916`

Deutung für normalverteilte Zufallsvariablen:

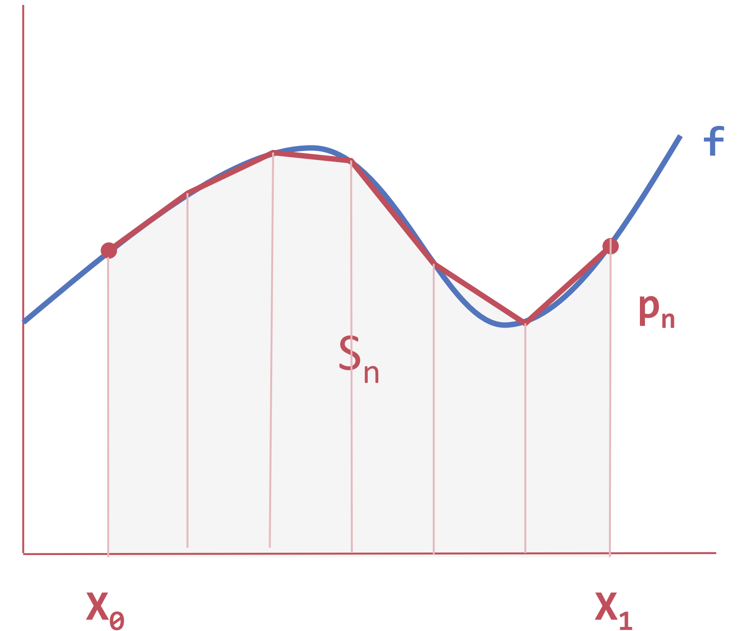
Innerhalb von 1 Standardabweichung liegen 68% der Daten

Innerhalb von 2 Standardabweichungen liegen 95% der Daten

Innerhalb von 3 Standardabwiechungen liegen 99.7% der Daten

Andere Integrationsregeln

Trapezregel: Stückweise affine Approximation




```
public class TrapezoidalIntegrator extends Integrator{
    TrapezoidalIntegrator(int pieces){
        n= pieces;
    }

    public double Integrate(Function f, double x0, double x1){
        double sum = 0;
        double width=(x1-x0)/n;
        for (int i = 0; i<n; ++i){
            double x = x0 + i*width;
            double y = f.Evaluate(x) + f.Evaluate(x+width);
            sum += y*width/2;
        }
        return sum;
    }
}
```

Beobachtung

Trapezregel ist für viele Funktionen kaum besser als die Rechteckregel.

Eine Inspektion des Algorithmus zeigt, dass Rechteckregel und Trapezregel auch nahezu dasselbe tun.

Für eine lineare Funktion stimmen die Regeln sogar überein.

Beispiel: Numerische Integration von sin(x) im Intervall $[0, \pi]$

n	Rechteck	Trapez
1	3.14159	0
2	2.22144	1.57079
4	2.05234	1.89611
8	2.01290	1.97423
16	2.00321	1.99357
32	2.00080	1.99839
64	2.00020	1.99960
128	2.00005	1.99990

Nebenbemerkung

nicht prüfungsrelevant

Betrachtung eines Einzelstückes der numerischen Integration am Intervall $[l, r]$.

Annahme: Funktion f genügend oft differenzierbar

Taylor-Entwicklung um $\tilde{x} = \frac{l+r}{2}$

$$f(x) = f(\tilde{x}) + (x - \tilde{x})f'(\tilde{x}) + \frac{(x-\tilde{x})^2}{2}\mathrm{f}''(\tilde{x}) + \frac{(x-\tilde{x})^3}{6}\mathrm{f}^{(3)}(\tilde{x}) + \frac{(x-\tilde{x})^4}{24}\mathrm{f}^{(4)}(\tilde{x}) + \ \dots$$

Integration von f ergibt (mit $\Delta = r - l$)

$$\int_l^r f(x)dx = \Delta \cdot f(\tilde{x}) + \frac{\Delta^3}{24} \cdot f''(\tilde{x}) + O(\Delta^5)$$

Nebenbemerkung

nicht prüfungsrelevant

Seien $I(f)$ das exakte Integral, $I^R(f)$ und $I^T(f)$ die Approximationen mit Rechteck- / Trapezregel.

Dann gelten

$$I^R(f) = I(f) - \frac{\Delta^3}{24} f''(\tilde{x}) + O(\Delta^5)$$

$$I^T(f) = I(f) + \frac{\Delta^3}{12} f''(\tilde{x}) + O(\Delta^5)$$

Der Fehler ist immerhin mit der dritten Potenz der Länge der Stücke kontrollierbar

Das verhilft zu folgendem **Trick**

$$2 \cdot I^R(f) + I^T(f) = 3 \cdot I(f) + O(\Delta^5)$$

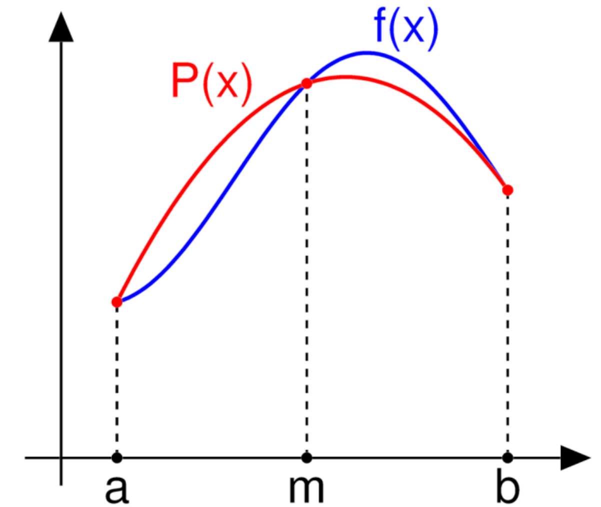
Noch besser: Der Fehler fällt mit der 5ten Poztenz!!

Daraus folgt die **Simpson Regel**

$$I(f) \approx I^S(f) := \frac{r-l}{6}\left(f(x_l) + 4f\left(\frac{r+l}{2}\right) + f(x_r)\right)$$

Simpson Regel

Simpson-Integration entspricht quadratischer Interpolation von f



Quadratische Interpolation
Quelle: Wikipedia

```
public class SimpsonIntegrator extends Integrator{
    SimpsonIntegrator(int pieces){
        n= pieces;
    }

    public double Integrate(Function f, double x0, double x1) {
        double sum = 0;
        double width=(x1-x0)/n;
        for (int i = 0; i<n; ++i)
        {
            double x = x0 + i*width;
            double y = f.Evaluate(x) +f.Evaluate(x+width) + 4*f.Evaluate(x+width/2);
            sum += y*width/6;
        }
        return sum;
    }
}
```

NB: Newton Cotes Formeln

nicht prüfungsrelevant

Die Newton-Cotes Formeln sind numerische Quadraturformeln zur approximativen Berechnung von Integralen durch Interpolation von Funktionen mit Lagrange Polynomen.

Rechteck-, Trapez- und Simpson-Regeln sind Beispiele dieser Formeln.

Eine andere berühmte numerische Integrationsmethode ist die Gauss-Quadratur

Darüber hinaus sind Verfahren interessant, die das Abtastgitter in x-Richtung adaptiv wählen. Solche Verfahren sind wichtig bei Funktionen, die nicht überall gleich «glatt» sind.

Für die Integration von Funktionen in einem hochdimensionalen Raum sind probabilistische Verfahren (Sampling) interessant.

Experiment: Vergleich der Verfahren

Approximation des Integrals $\int_0^\pi \sin(x)\,dx = 2$

n	Rechteck	Trapez	Simpson
1	3.14159265	0	2.0943951
2	2.22144147	1.57079633	2.00455975
4	2.05234431	1.8961189	2.00026917
8	2.01290909	1.9742316	2.00001659
16	2.00321638	1.99357034	2.00000103
32	2.00080342	1.99839336	2.00000006
64	2.00020081	1.99959839	2
128	2.0000502	1.9998996	2
256	2.00001255	1.9999749	2
512	2.00000314	1.99999373	2
1024	2.00000078	1.99999843	2

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Kapselung, Information Hiding

- Verbergen des Zustands und der Implementierungsdetails von Objekten
- Definition einer Schnittstelle zum Zugriff auf interne Datenstruktur → Abstraktion
- Ermöglicht das Sicherstellen von Invarianten

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Vererbung

- Objekte können Eigenschaften von Objekten erben
- Abgeleitete Objekte können neue Eigenschaften besitzen oder vorhandene überschreiben
- Macht Code- und Datenwiederverwendung möglich

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Polymorphie

- Ein Bezeichner kann abhängig von seiner Verwendung unterschiedliche Datentypen annehmen.
- Unterschiedliche Datentypen können bei gleichem Zugriff auf ihr gemeinsames Interface verschieden reagieren.
- Macht „nicht invasive” Erweiterung von Bibliotheken möglich.