

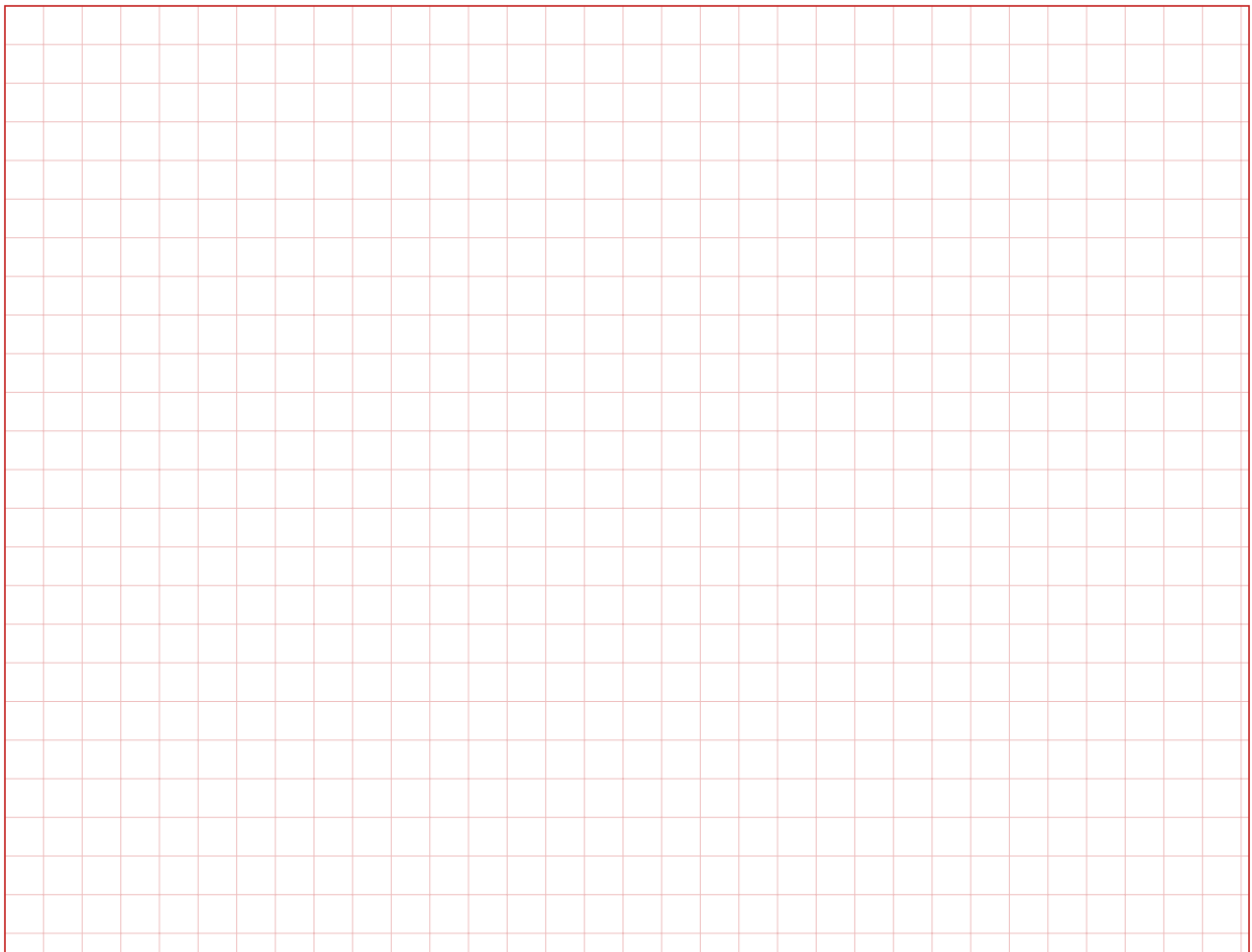
# Programmieren II: Java

## 4. Praktikum (Abgabe 21. Juni, 24 Uhr)

Auer, Hanel, Jürgensen, Lehner, Riemenschneider



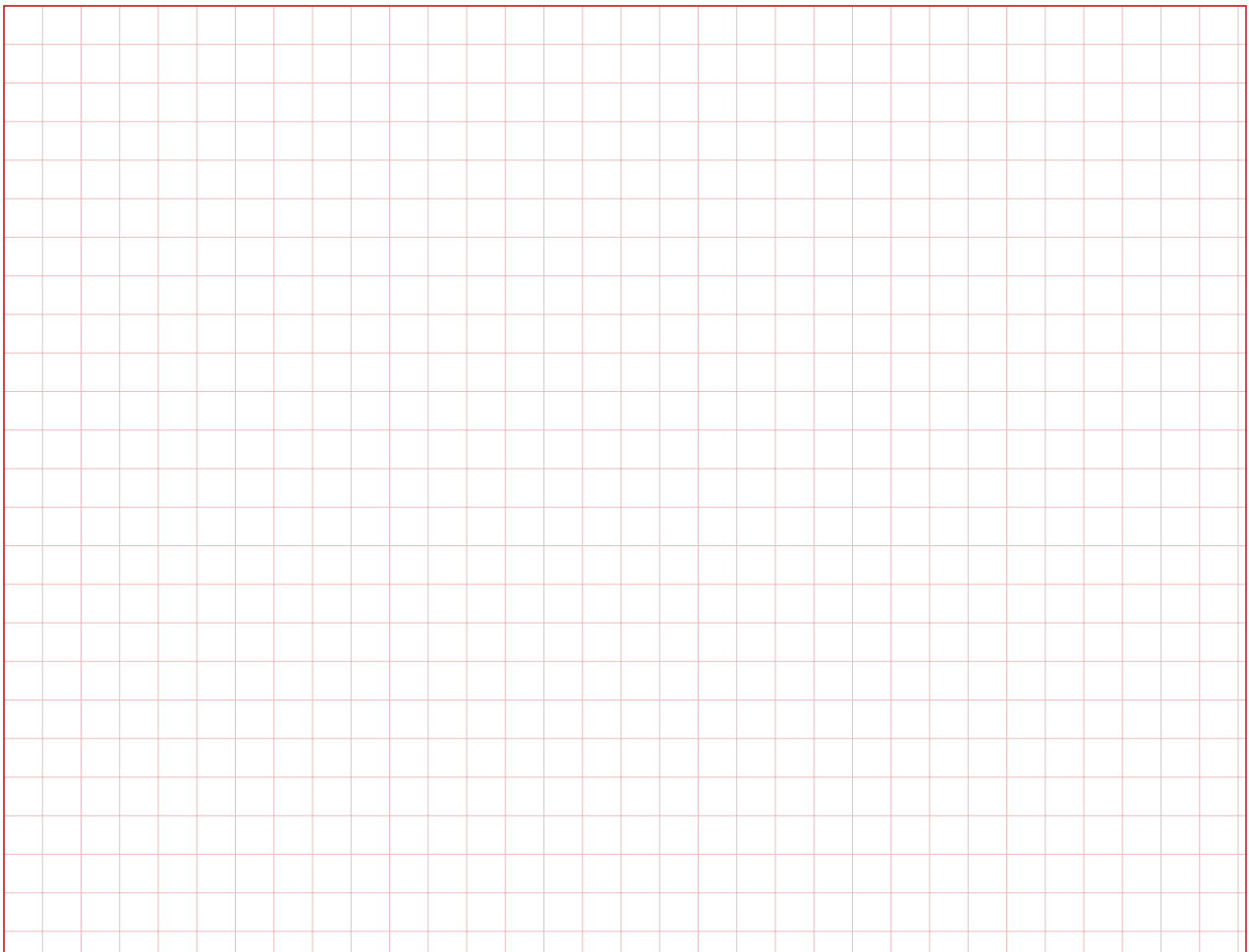
### Notizen



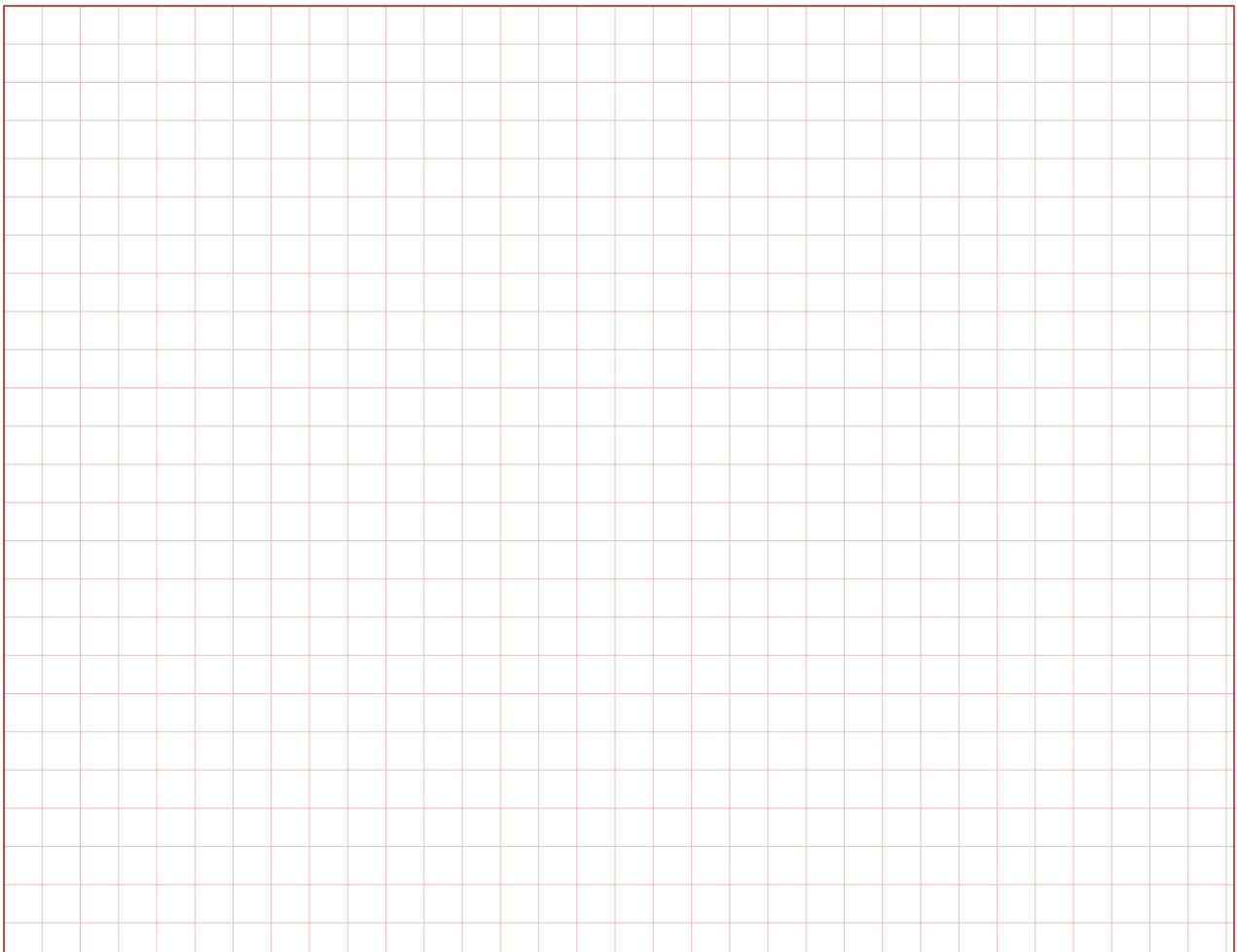
Lernziele

Reelle Funktionen

Notizen



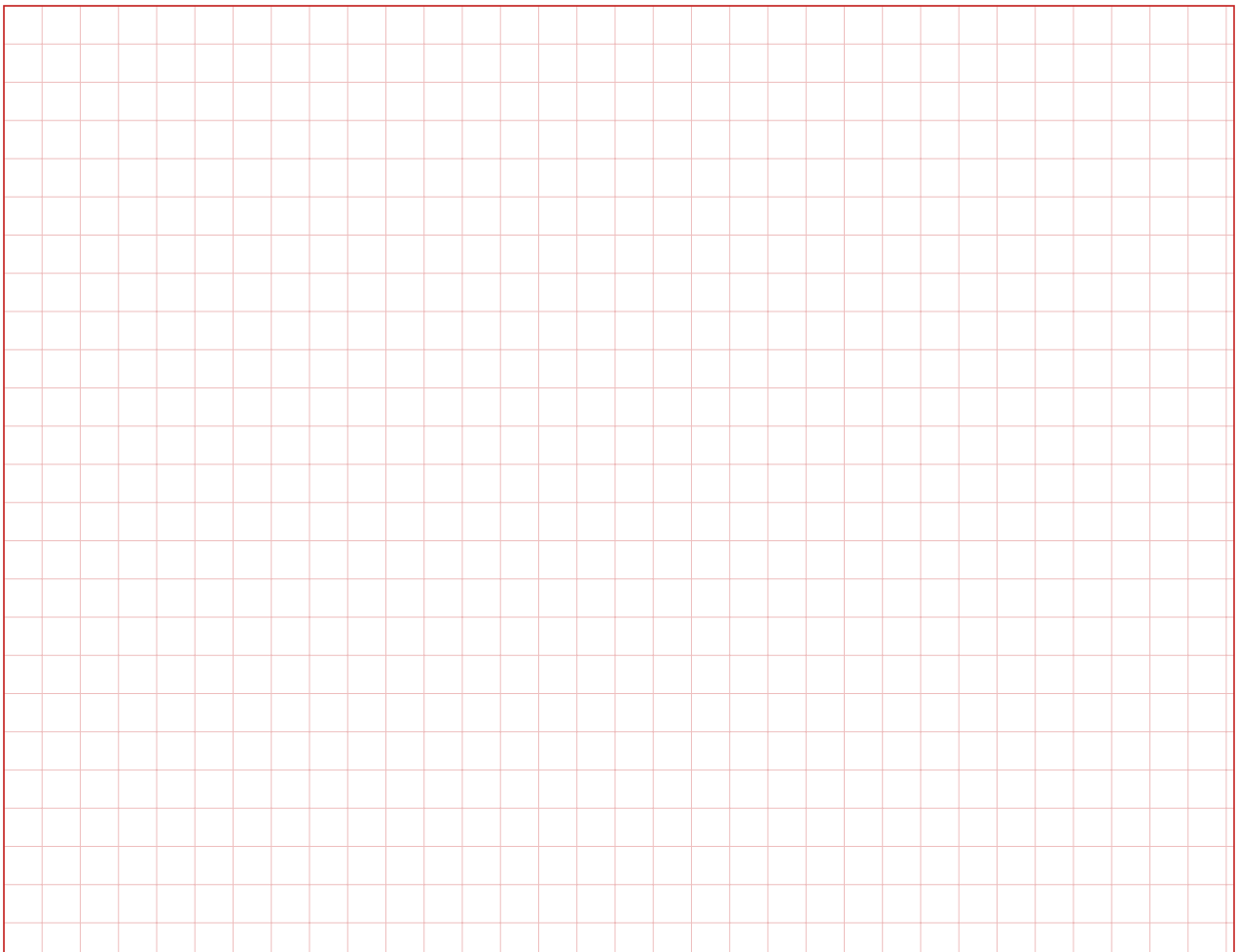
### Notizen



# Hinweise zum Praktikum

- ▶ **Abgabetermin** dieses Praktikums: **Abgabe 21. Juni, 24 Uhr**
- ▶ Sie dürfen die Aufgaben **alleine** oder zu **zweit** abgeben
- ▶ **Kennzeichnen** Sie Ihre Abgabe entsprechend mit Ihren Namen
- ▶ Sie müssen **4 der 5** Praktika bestehen
- ▶ **Kommentieren** Sie Ihren Code
  - ▶ Jede **Methode** (wenn nicht vorgegeben)
  - ▶ **Wichtige** Anweisungen/Code-Blöcke
  - ▶ **Nicht kommentierter** Code führt zu **Nichtbestehen**
- ▶ Bestehen Sie eine Abgabe **nicht** haben Sie einen **zweiten Versuch**, in dem Sie Ihre Abgabe **verbessern müssen**
- ▶ **Wichtig**
  - ▶ Sie sind einer **Praktikumsgruppe** zugewiesen, **nur** in dieser werden Ihre Abgaben **akzeptiert**
  - ▶ Beachten Sie dazu die Hinweise auf der **Moodle-Kursseite**

## Notizen

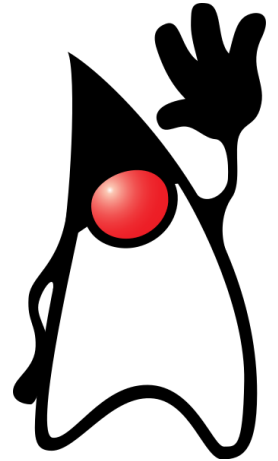


## Notizen

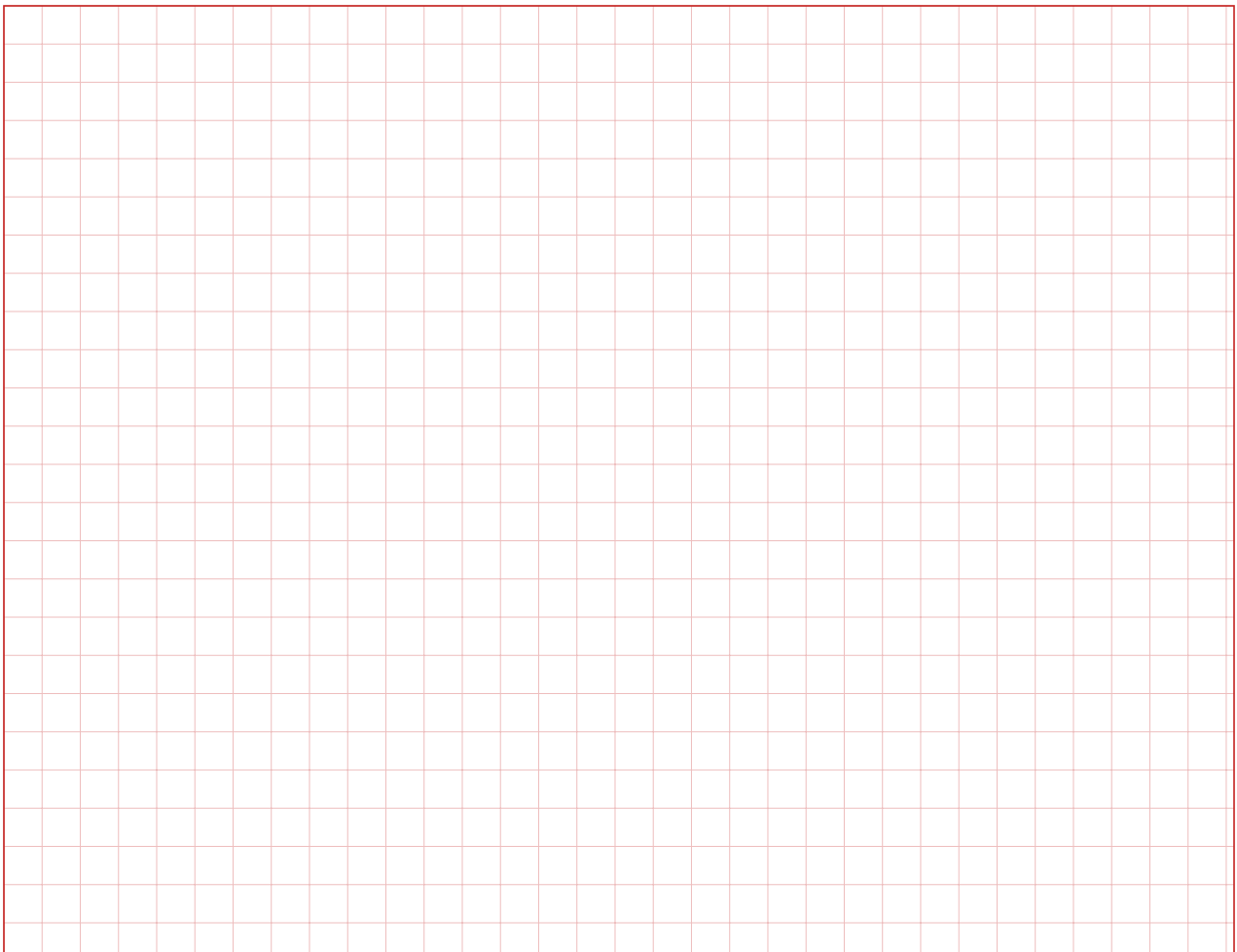
# Willkommen zum 4. Praktikum!

## Lernziele

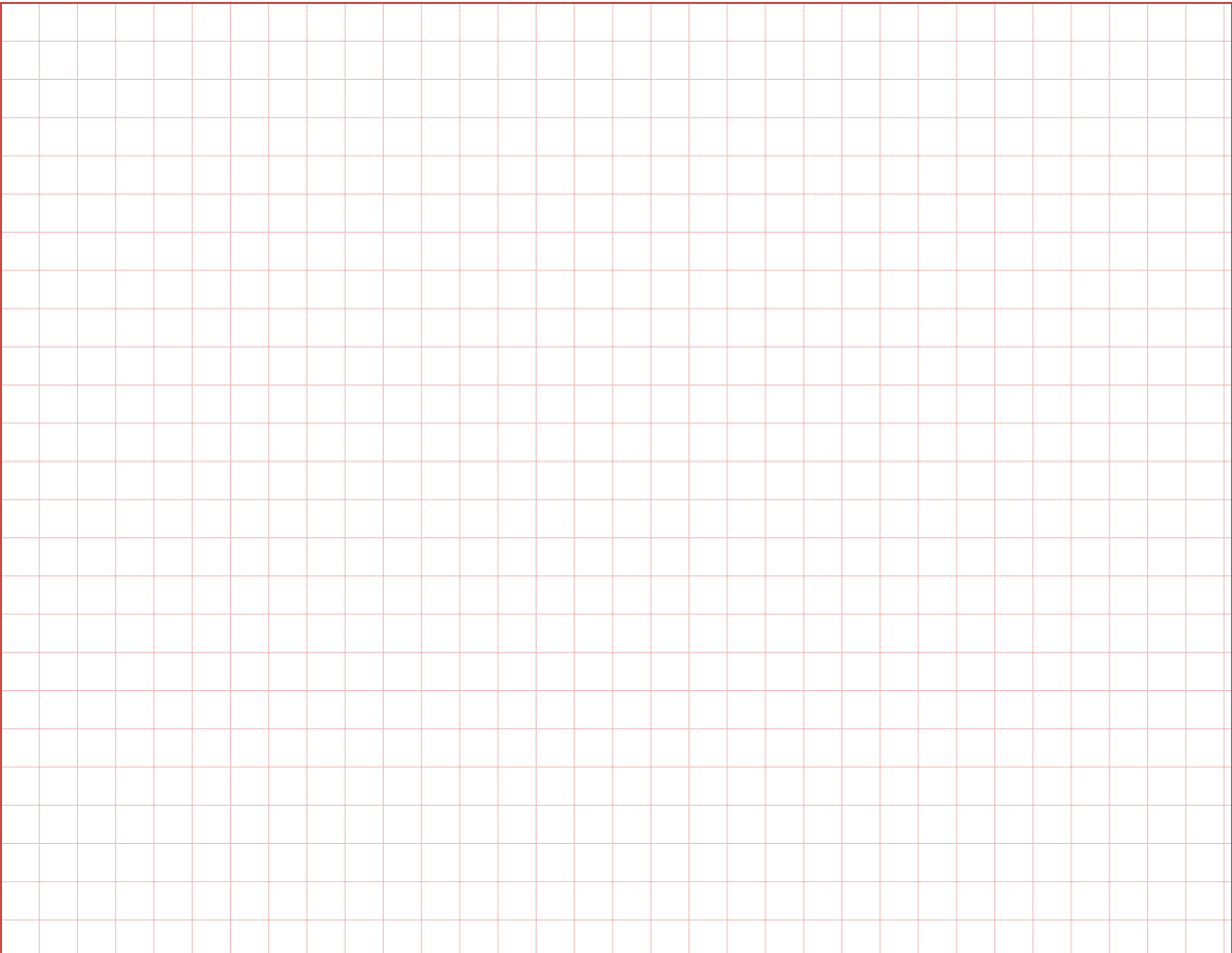
- ▶ **Abstrakte Klassen:** definieren, ableiten
- ▶ **Vererbung:** Methoden überschreiben, dynamische Bindung
- ▶ **Interfaces:** erstellen, implementieren
- ▶ **Methoden von Object überschreiben:** toString
- ▶ **Ausnahmen:** erzeugen, eigene Ausnahmetypen



## Notizen



Notizen



# Inhalt

## Reelle Funktionen

Einführung

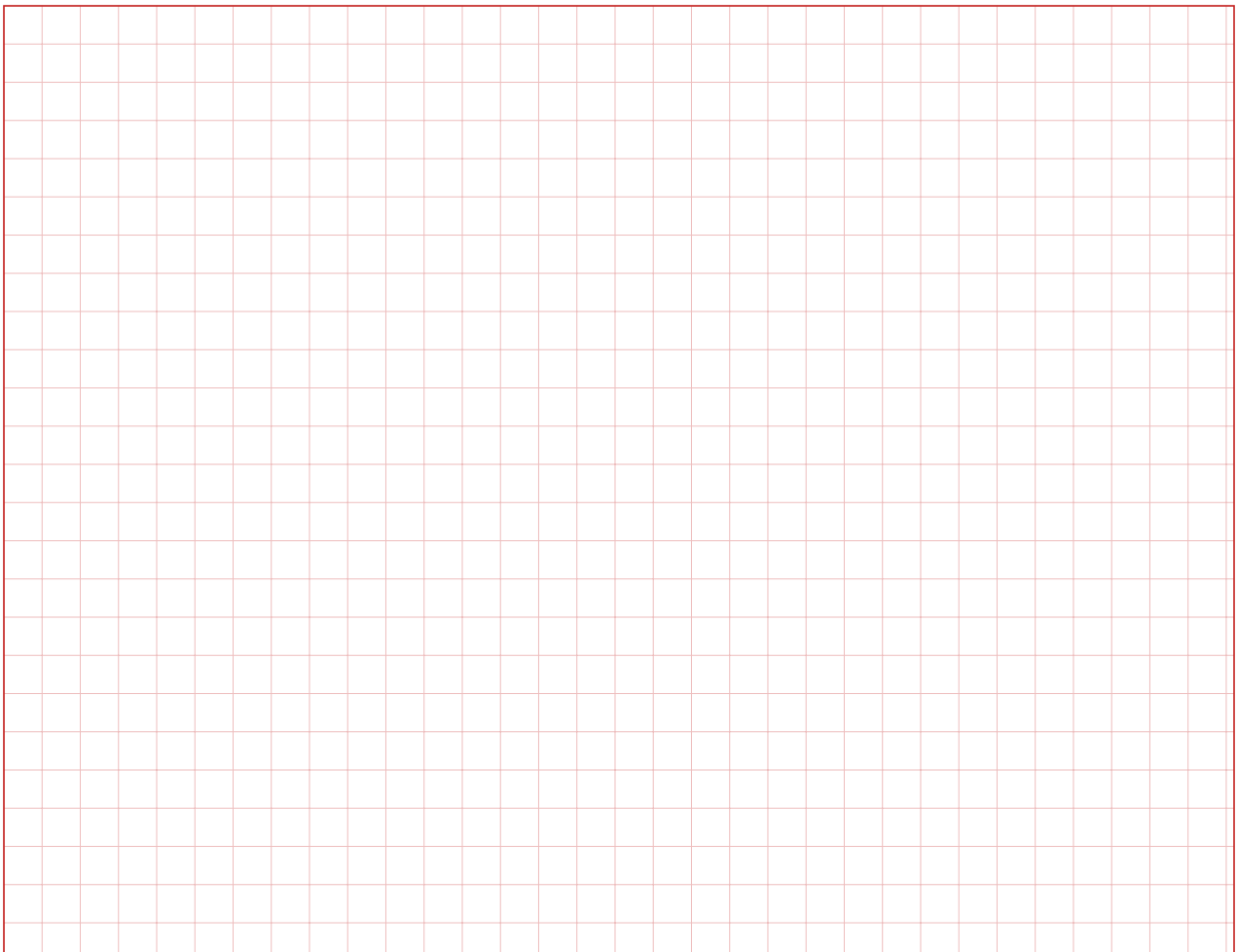
Die Ausnahmen zuerst

Klasse RealFunction

Kombination von Funktionen

Differenzierbare Funktionen

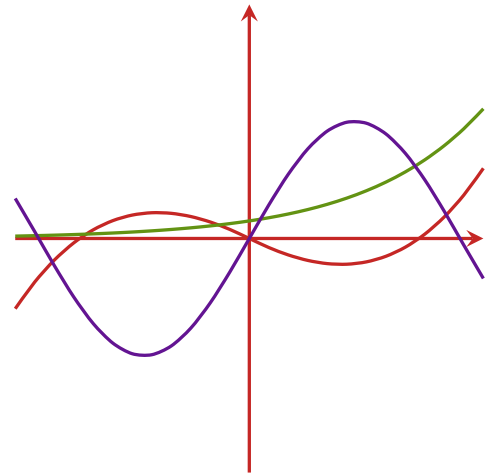
## Notizen



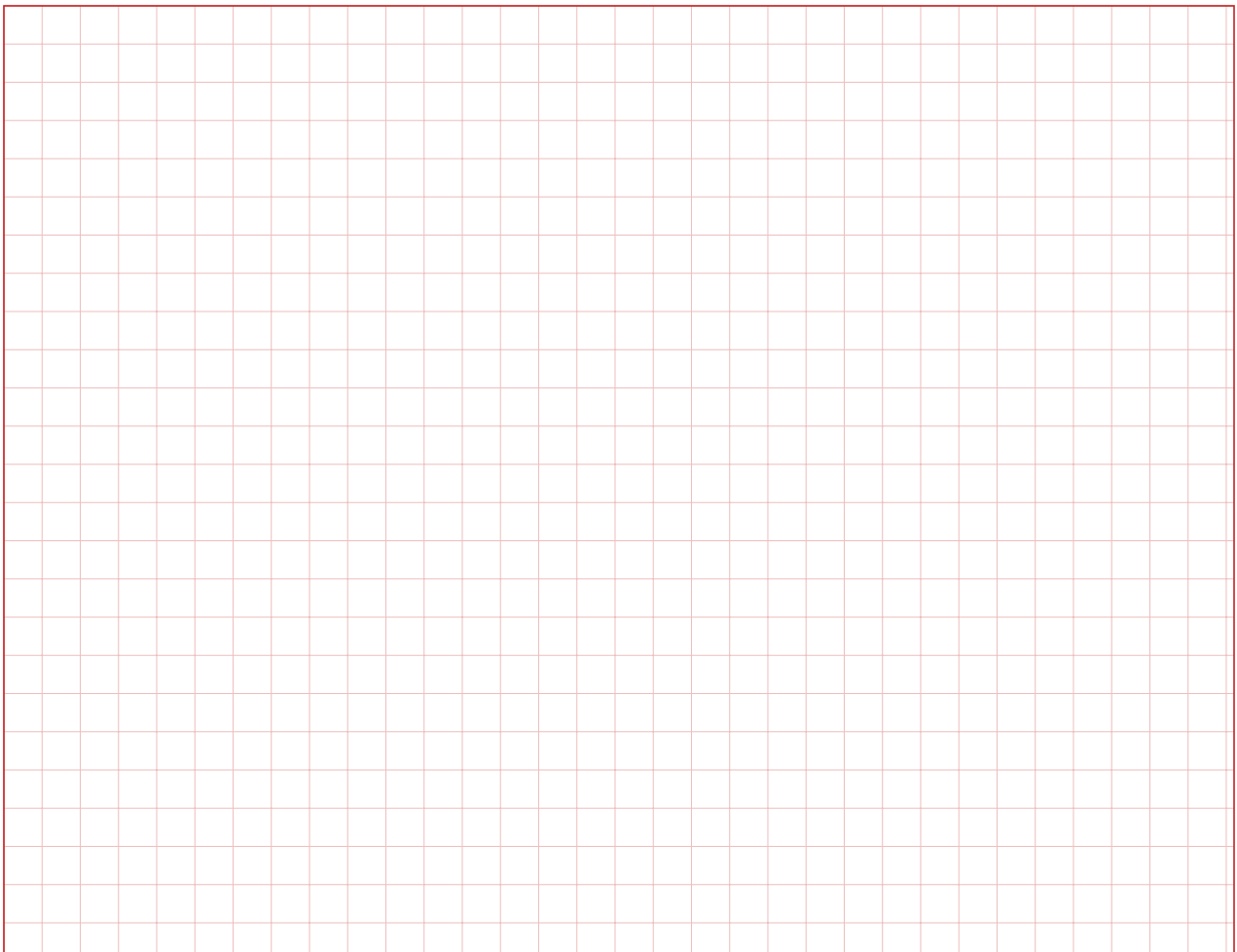


# Ein einfaches Analysis-Paket

- ▶ Diese Übung lehnt sich an die Vorlesung „Mathematik II (Analysis)“ an
- ▶ Wir modellieren in einer **Klassenbibliothek**
  - ▶ Reelle Funktionen (**abstrakte Klasse**)
  - ▶ Komposition von Funktionen (**Assoziationen**)
  - ▶ Differenzierbare Funktionen (**Interface**)



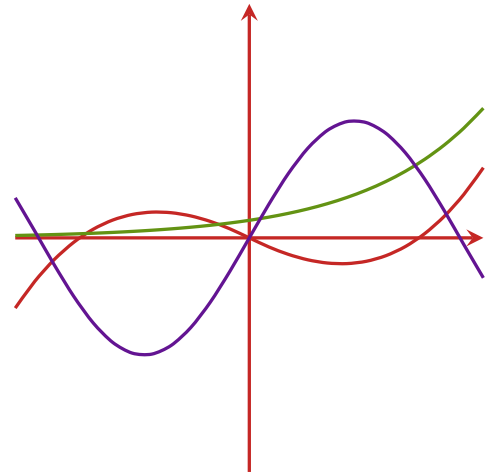
## Notizen



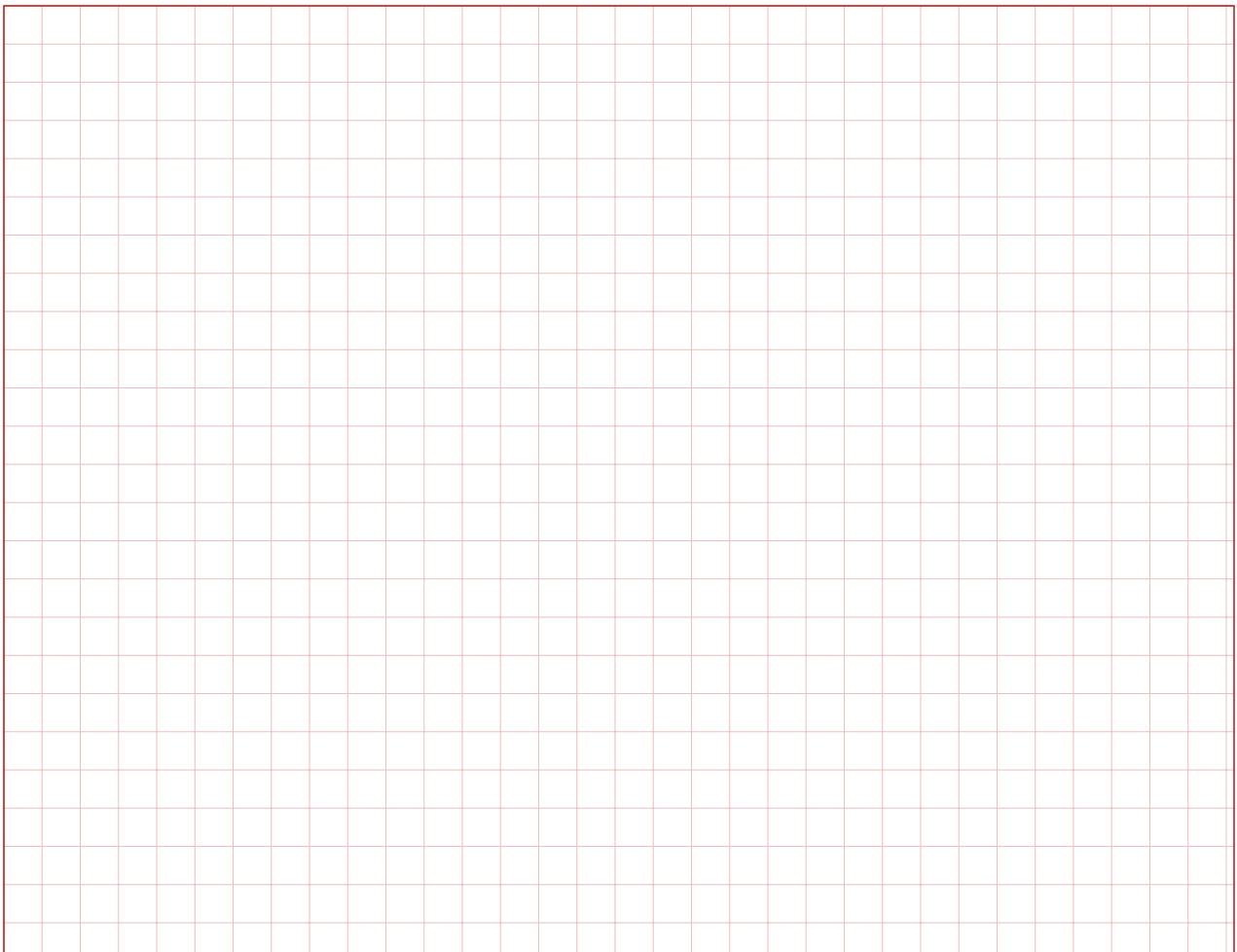
# Hinweise vorab

## ► Hinweise vorab

- **Keine Panik**: Die mathematischen Konzepte, die Sie für diese Übung brauchen, bleiben auf Schul-Niveau
- Platzieren Sie alle Klassen/Typen im **Paket** `de.hawlandshut.calculus`
  - Erstellen Sie dazu die entsprechende **Verzeichnisstruktur**
  - Versehen Sie alle Typen mit der **package-Deklaration**
- Für die einzelnen Klassen sind jeweils **JUnit-Tests** gegeben
- Implementieren sich nach eigenem Ermessen Programme um ihre Implementierung zu testen (**sehr empfohlen**)
- Sie können auch die **Plotter**-Klasse zum Plotten Ihrer Funktionen verwenden (s. unten)



## Notizen



## Reelle Funktionen

Einführung

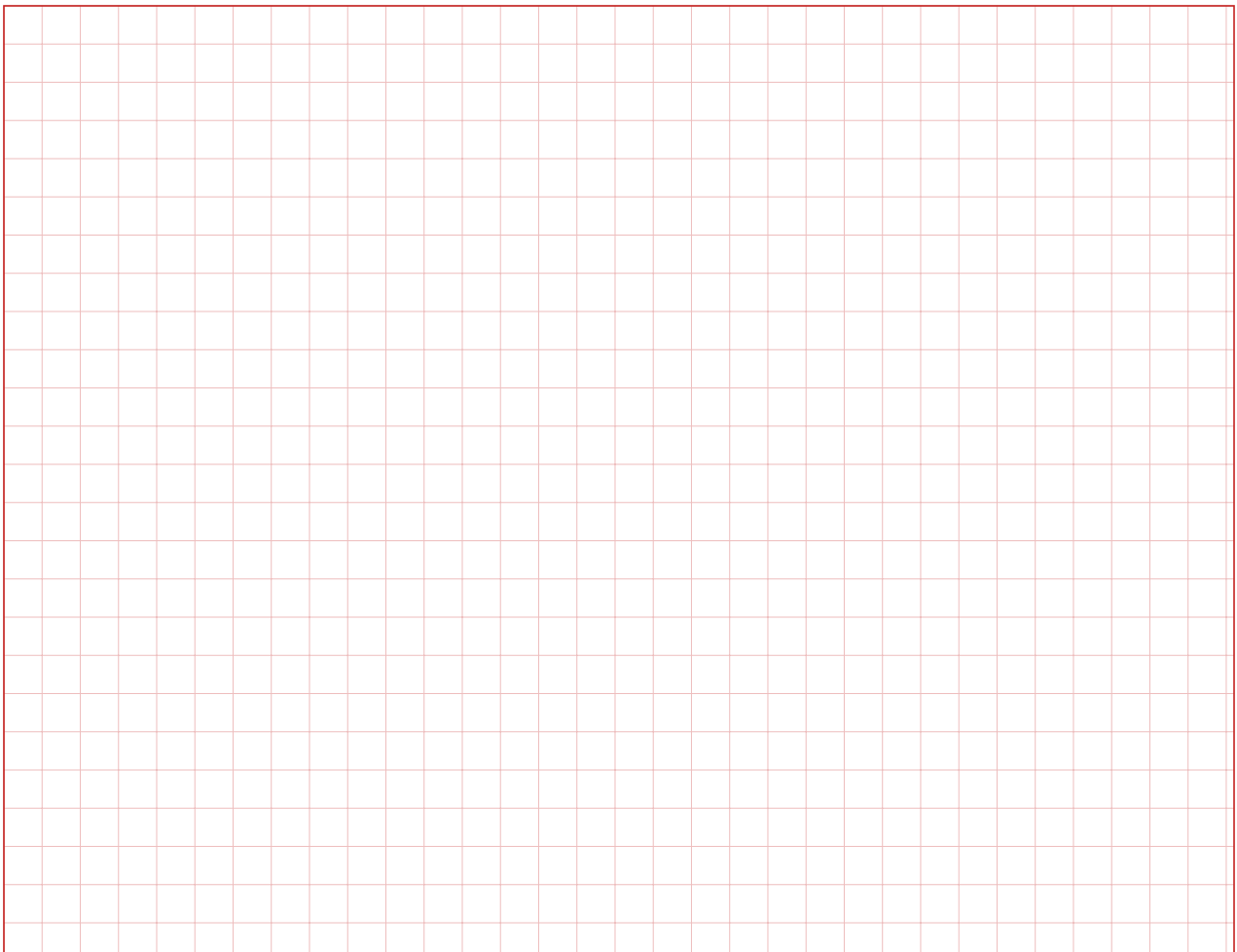
Die Ausnahmen zuerst

Klasse RealFunction

Kombination von Funktionen

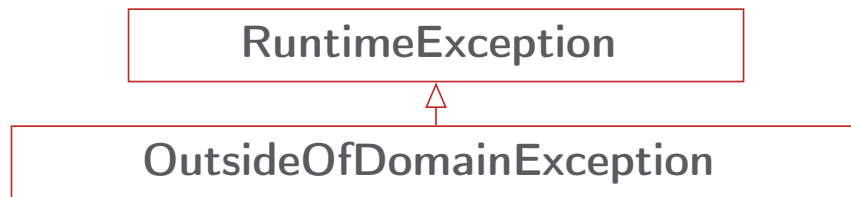
Differenzierbare Funktionen

## Notizen



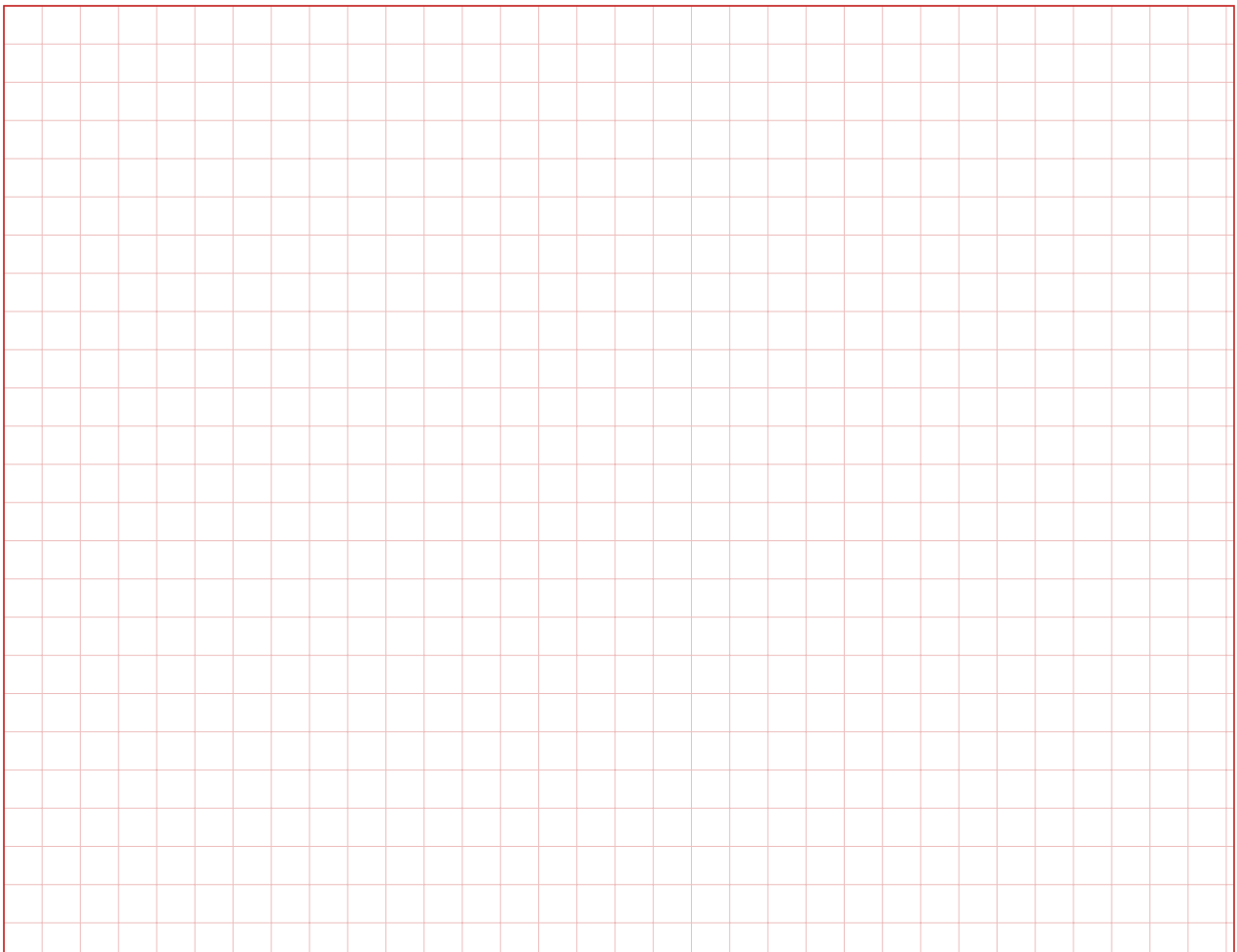
# Ausnahmen

- ▶ Wir benötigen für unsere Implementierung eigene Ausnahmen



- ✓ Implementieren Sie `OutsideOfDomainException`
- ✓ Definieren Sie für die Klasse die **drei Konstruktoren**, wie in den Vorlesungsfolien definiert
- ▶ Die **genaue Bedeutung** der Ausnahme wird im Verlauf der Übung klar

## Notizen



# Inhalt

## Reelle Funktionen

Einführung

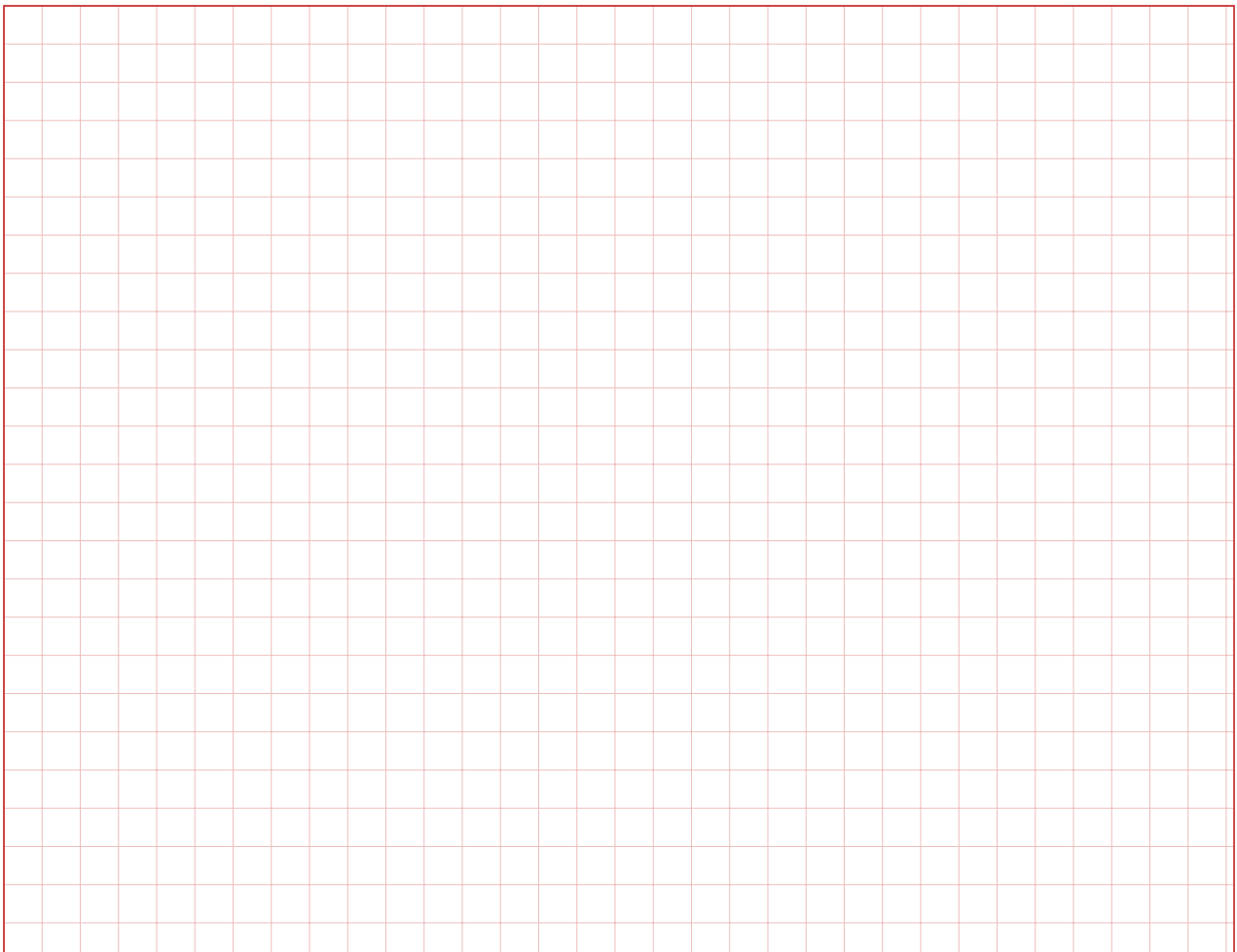
Die Ausnahmen zuerst

Klasse `RealFunction`

Kombination von Funktionen

Differenzierbare Funktionen

## Notizen

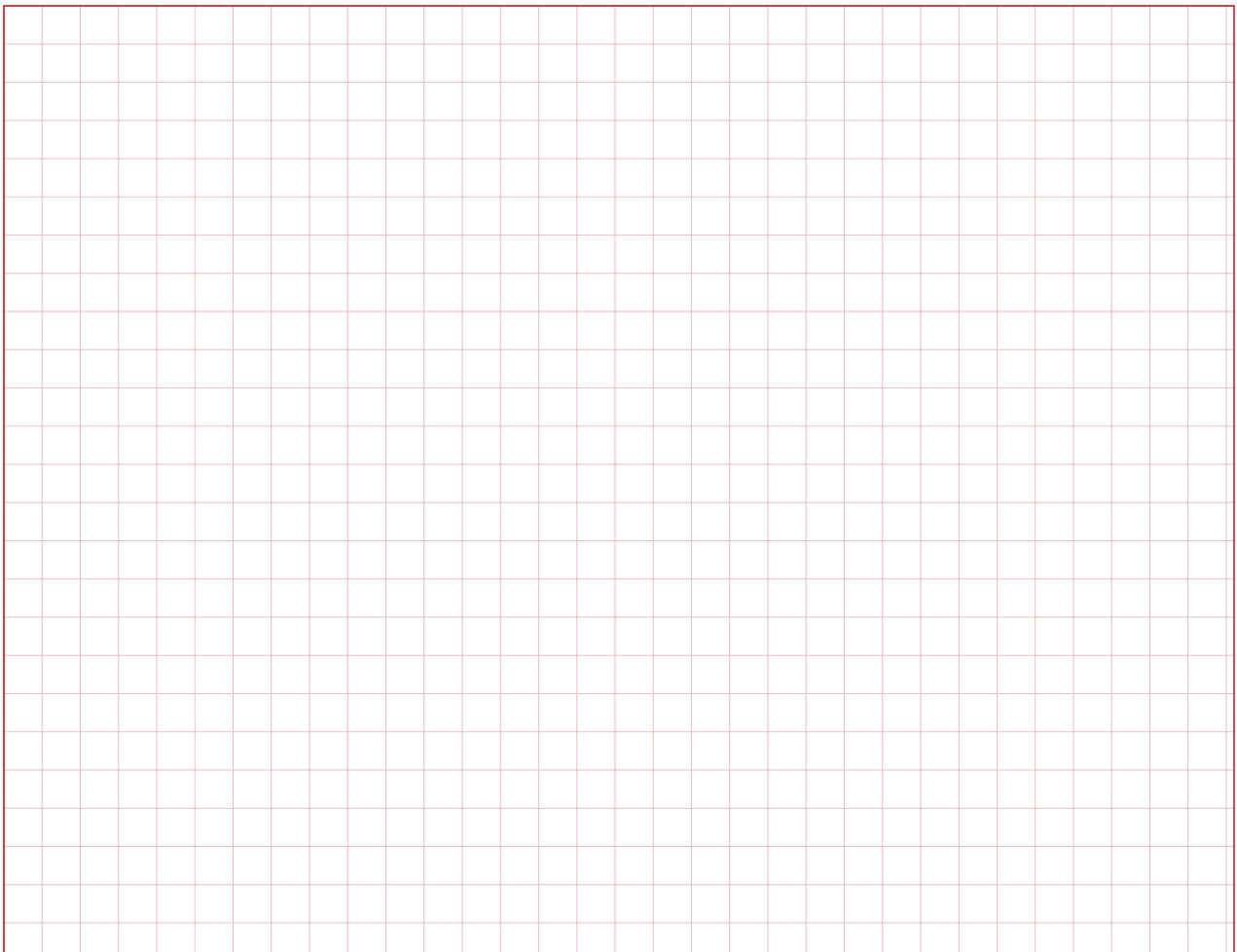


# Klasse RealFunction

<code>&lt;&lt;abstract&gt;&gt;</code> <b>RealFunction</b>
<code>+ evaluateAt(x : double) : double</code> <code>+ inDomain(x : double) : boolean</code>

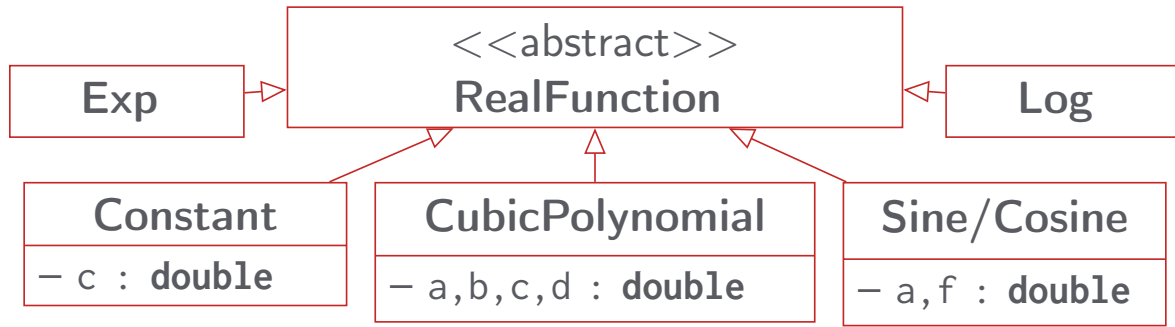
- ▶ Die **abstrakte Klasse** modelliert **reelle Funktionen**  $f : \mathbb{R} \rightarrow \mathbb{R}$
  - ▶ `evaluateAt`
    - ▶ **abstract**
    - ▶ Funktionswert an der Stelle `x`
    - ▶ Erzeugt `OutsideOfDomainException` wenn `x` außerhalb des **Definitionsbereichs** ist (engl. „domain“)
  - ▶ `inDomain`
    - ▶ **abstract**
    - ▶ **true** wenn `x` im Definitionsbereich liegt, sonst **false**
- ✓ **Implementieren** Sie `RealFunction`!

## Notizen



# Konkrete Funktionen

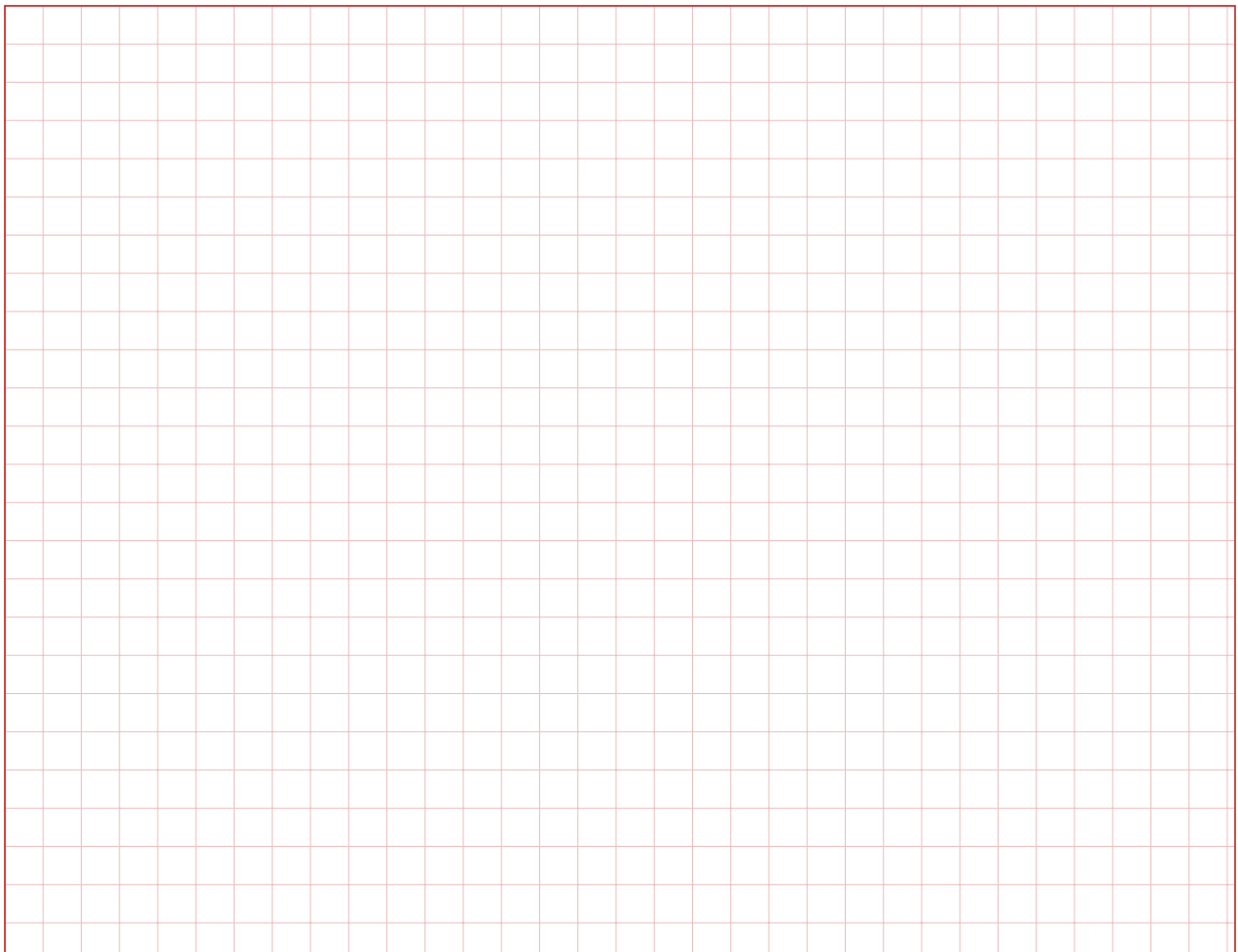
Folgende (konkrete) Klassen leiten von RealFunction ab



Klasse	Definition	Java-Fkt.	Def.b.
Constant	$c$	—	$\mathbb{R}$
CubicPolynomial	$ax^3 + bx^2 + cx + d$	—	$\mathbb{R}$
Sine/Cosine	$a \cdot \sin / \cos(f \cdot x)$	Math.sin/cos	$\mathbb{R}$
Exp	$e^x$	Math.exp	$\mathbb{R}$
Log	$\ln(x)$	Math.log	$x > 0$

**Hinweis:** Mit Sine/Cosine sind **zwei** Klassen gemeint

## Notizen



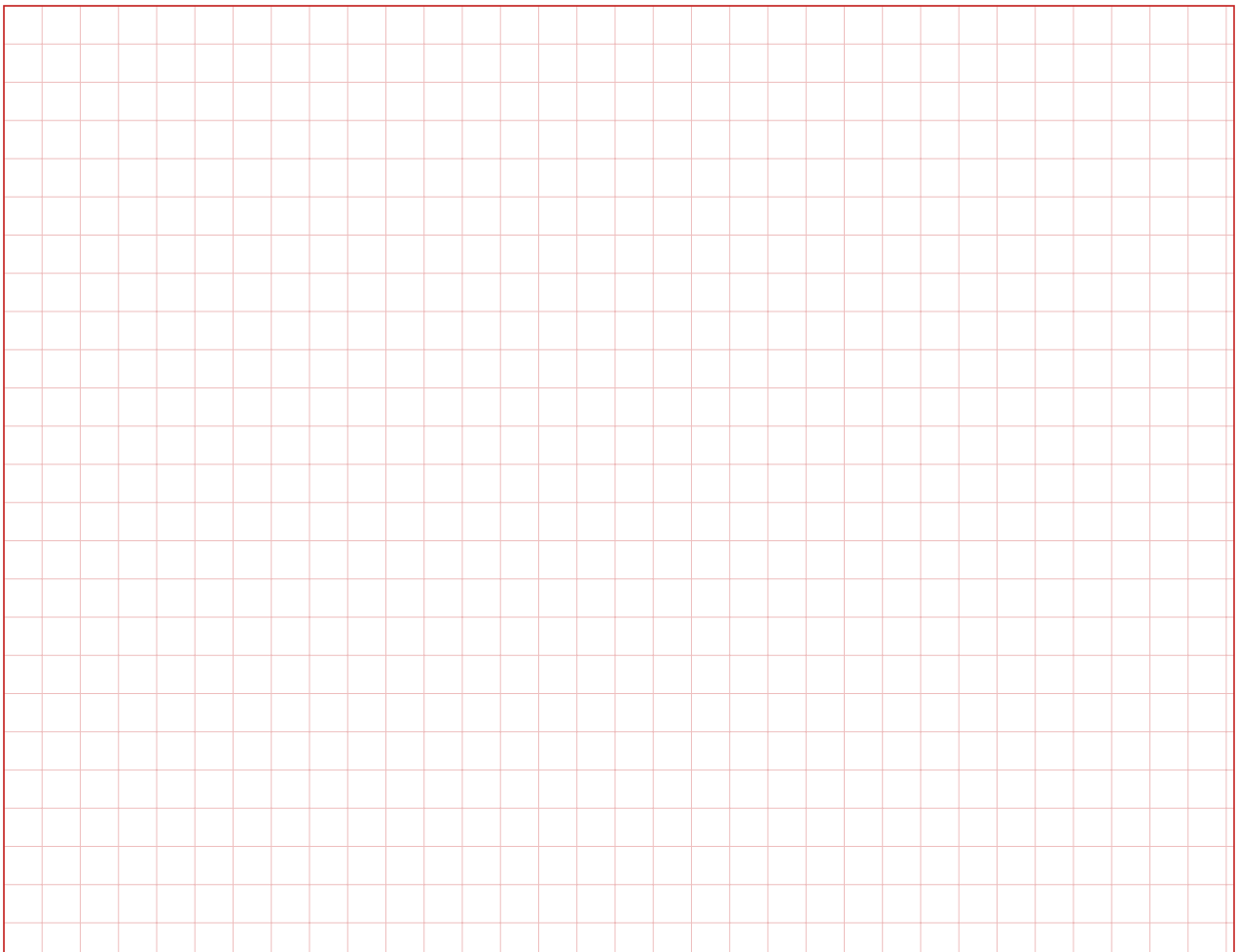
# Konkrete Funktionen

- ✓ Implementieren Sie die vorher definierten Klassen
  - ▶ Deklarieren Sie die Attribute (wenn vorhanden) **final**
  - ▶ Implementieren Sie jeweils einen **Konstruktor**, der die Attribute (wenn vorhanden) **initialisiert**
  - ▶ Generieren Sie eine `OutOfDomainException` falls nötig
  - ▶ Überschreiben Sie die Methode `toString` wie folgt

Klasse	Rückgabe <code>toString()</code>
Constant	<code>c</code>
CubicPolynomial	<code>a*x^3+b*x^2+c*x+d</code>
Sine	<code>a*sin(f*x)</code>
Exp	<code>exp(x)</code>
Log	<code>ln(x)</code>

- ▶ Für die Parameter `a, b, c, d, f` setzen Sie die konkreten Werte über die **Formatanweisung** `%f` ein
- ▶ **Beispiel:** `sin.toString()` liefert `1.0*sin(2.0*x)`
- ▶ Verwenden Sie zum Testen `calculus/RealFunctionTest.java`

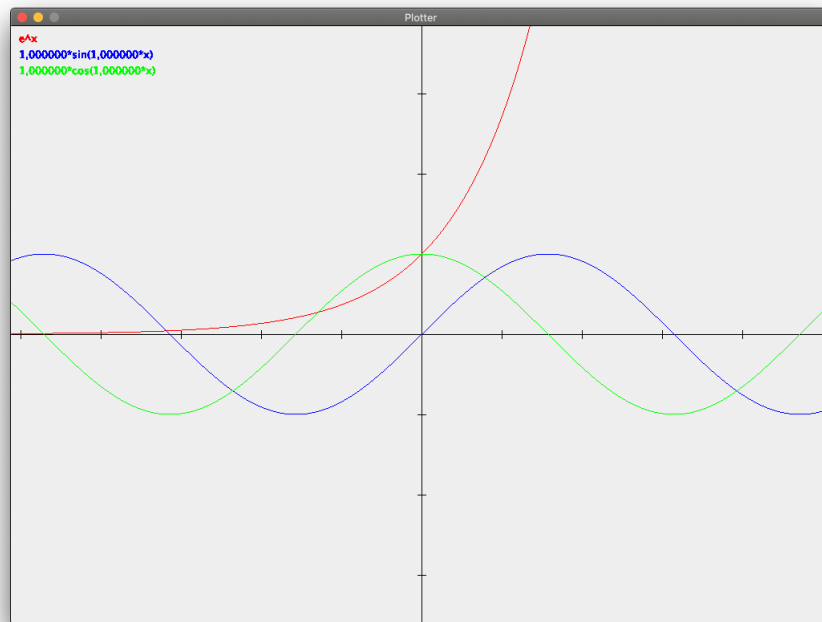
## Notizen



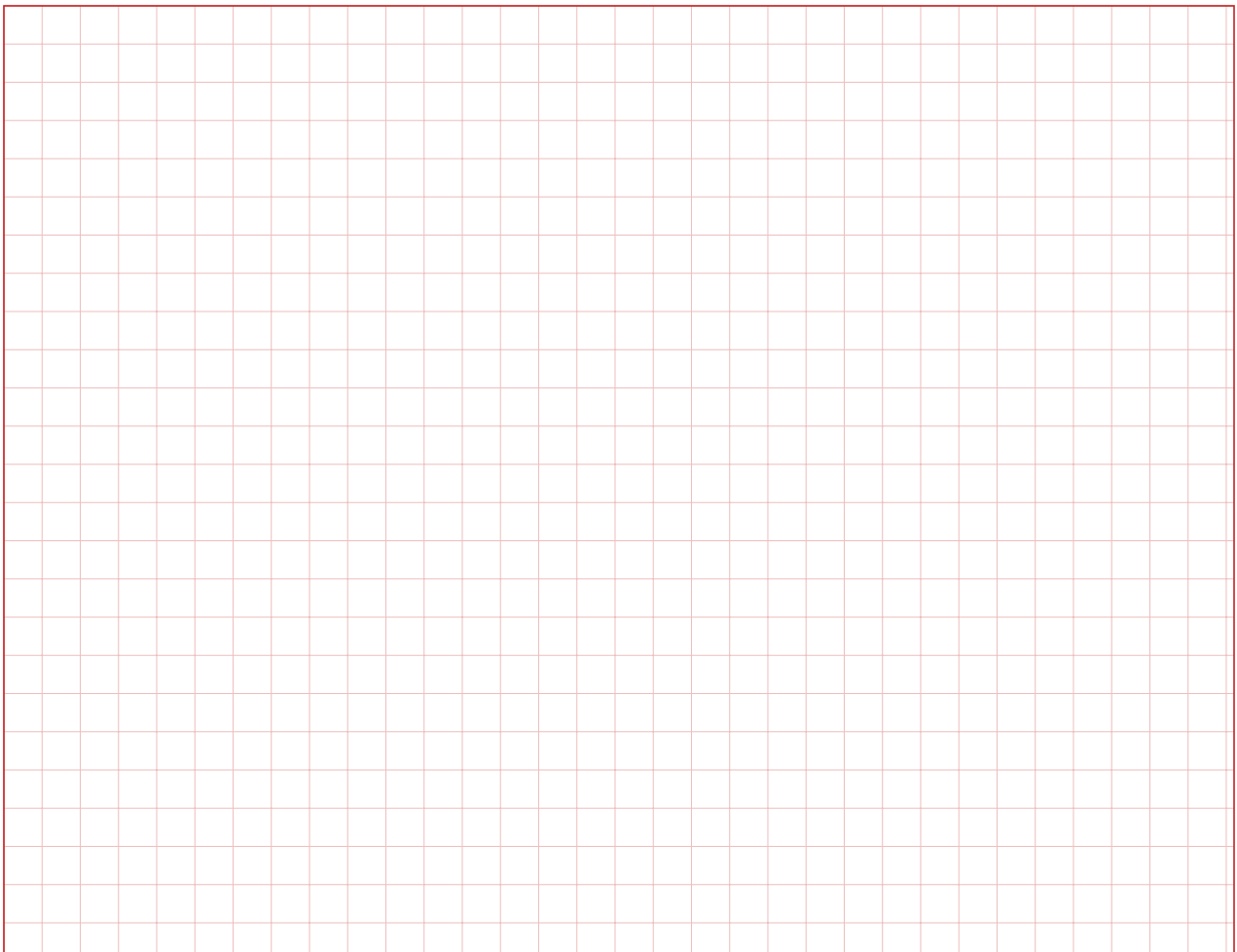


# Testen mit Plotter

- ▶ In `calculus/Plotter.java` ist ein **sehr einfacher** Plotter mit dem Sie Ihre Implementierungen testen können
  - ▶ **static void** `Plotter.plot(RealFunction... functions)` plottet die übergebenen Funktionen
  - ▶ **Beispiel:** `Plotter.plot(new Exp(), new Sine(1,1), ↵  
new Cosine(1,1))`



## Notizen



# Inhalt

## Reelle Funktionen

Einführung

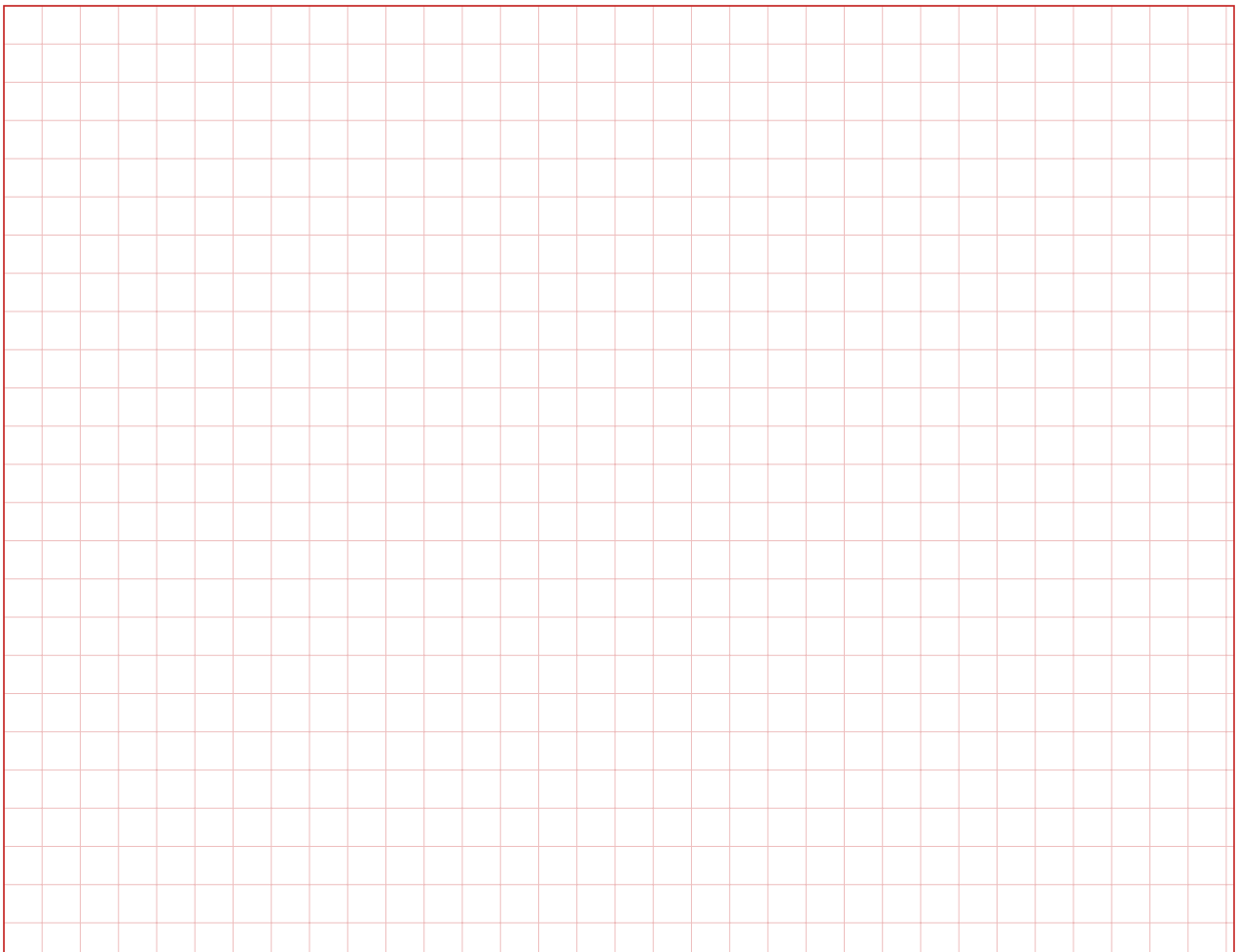
Die Ausnahmen zuerst

Klasse RealFunction

**Kombination von Funktionen**

Differenzierbare Funktionen

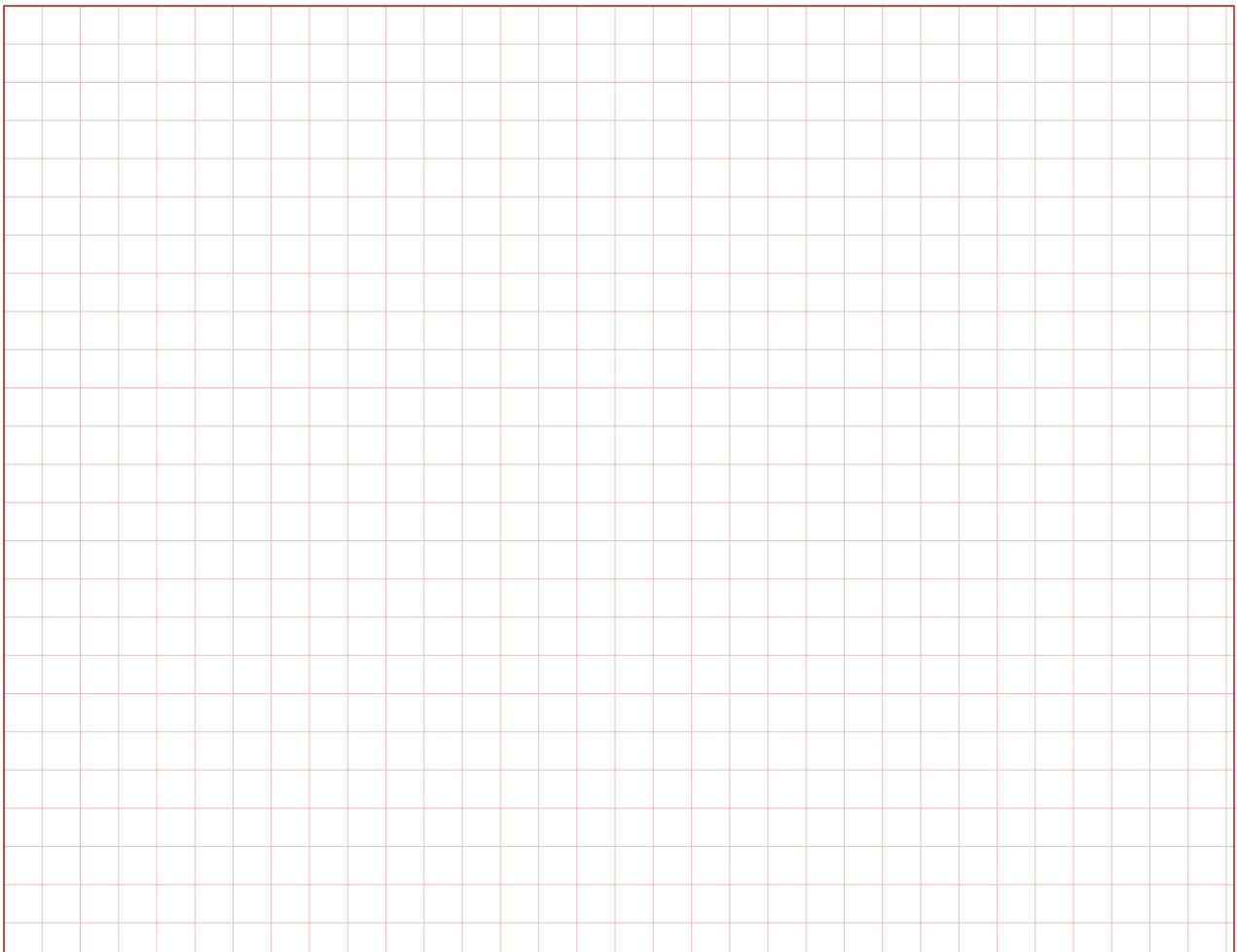
## Notizen



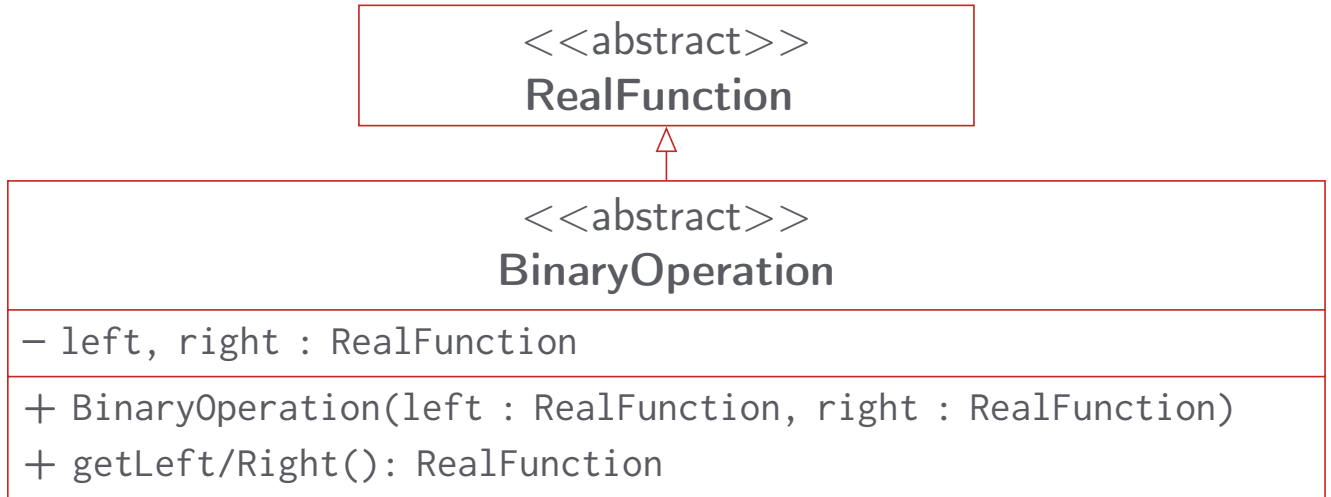
# Kombination von Funktionen

- ▶ Man kann komplexere Funktionen aus anderen Funktionen mit Hilfe **binärer Operationen zusammenbauen**
- ▶ Beispiele
  - ▶ **Addition** zweier Funktionen  $f + g$ 
    - ▶ **Linker Operand:**  $\sin(x)$
    - ▶ **Rechter Operand:**  $x^2 + 2x + 1$
    - ▶ **Addition:**  $(\sin(x)) + (x^2 + 2x + 1)$
  - ▶ **Multiplikation** zweier Funktionen  $f \cdot g$ 
    - ▶ **Linker Operand:**  $e^x$
    - ▶ **Rechter Operand:**  $x^3$
    - ▶ **Multiplikation:**  $e^x \cdot x^3$
  - ▶ **Komposition** zweier Funktionen  $f \circ g$ 
    - ▶ **Linker Operand:**  $x^2 - 2x$
    - ▶ **Rechter Operand:**  $\ln(x)$
    - ▶ **Komposition** **setzt** die rechte Funktion in die linke Funktion **ein**
    - ▶ **Komposition:**  $(\ln(x))^2 - 2(\ln(x))$

## Notizen



# Klasse BinaryOperation

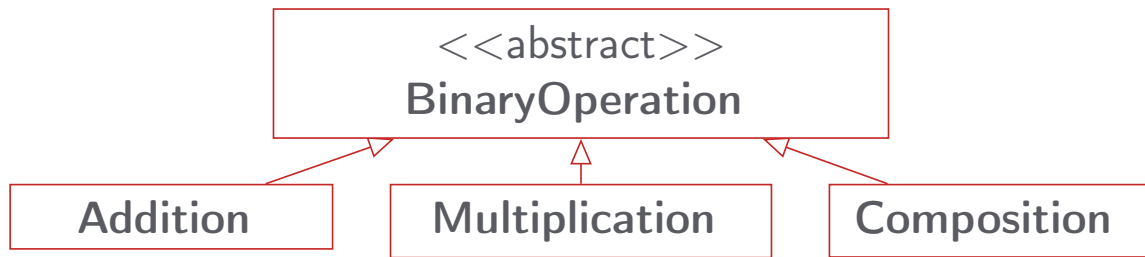


- ▶ BinaryOperation ist die **abstrakte** Basisklasse für die binären Operationen der vorherigen Folie
  - ▶ `left`, `right` sind die Operanden (**final**) mit Gettern
  - ▶ BinaryOperation definiert einen Konstruktor, der `left` und `right` initialisiert (jeweils `!= null`)
- ☑ Implementieren Sie BinaryOperation!

## Notizen

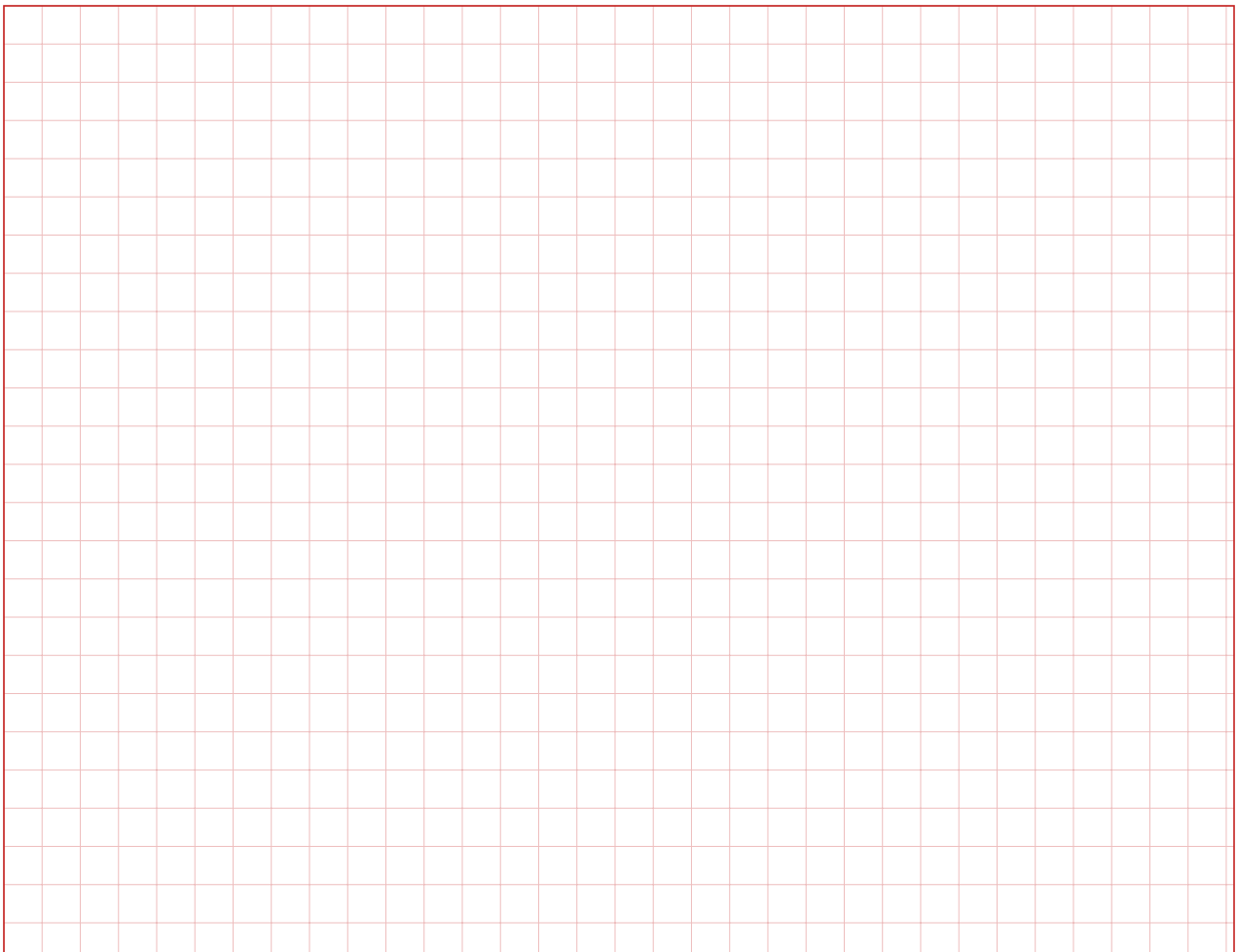
A large rectangular area with a light gray grid pattern, intended for students to take notes during the lecture.

# Konkrete binäre Operationen



Klasse	Operation	toString()
Addition	$f(x) + g(x)$	(left)+(right)
Multiplication	$f(x) * g(x)$	(left)*(right)
Composition	$(f \circ g)(x) = f(g(x))$	(left)o(right)

## Notizen



# Konkrete binäre Operationen

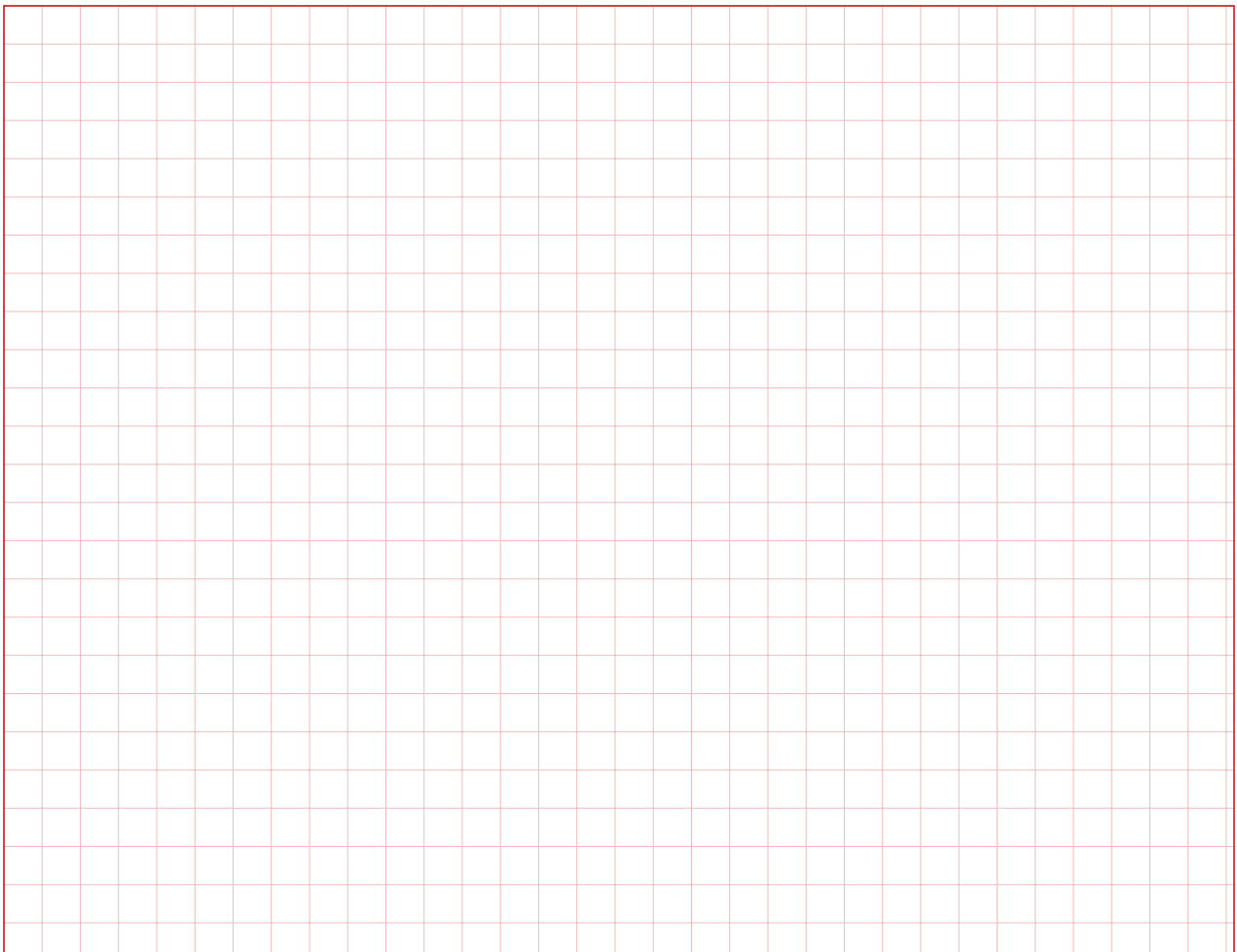
Implementieren Sie die angegebenen Operationen

- ▶ Rufen Sie in den Konstruktoren den Konstruktor von `BinaryOperation` auf
- ▶ Überschreiben Sie `inDomain(x)` mit
  - ▶ Addition/Multiplication:  $x$  muss im Definitionsbereich von `left` und `right` liegen
  - ▶ Composition:  $x$  muss im Definitionsbereich von `right` liegen und `right.evaluateAt(x)` im Definitionsbereich von `left`
- ▶ Überschreiben Sie `evaluateAt` wie oben definiert
- ▶ Überschreiben Sie `toString` wie oben definiert. `left` und `right` sollen dabei durch `getLeft().toString()` und `getRight().toString()` ersetzt werden. **Beispielausgabe:**

```
var left = new Sine(1,1); // toString: "1.0*sin(1.0*x)"
var right = new Exp(); // toString: "exp(x)"
var add = new Addition(left, right);
add.toString(); // "(1.0*sin(1.0*x))+(exp(x))"
```

- ▶ Verwenden Sie zum Testen `calculus/BinaryOperationTest.java`

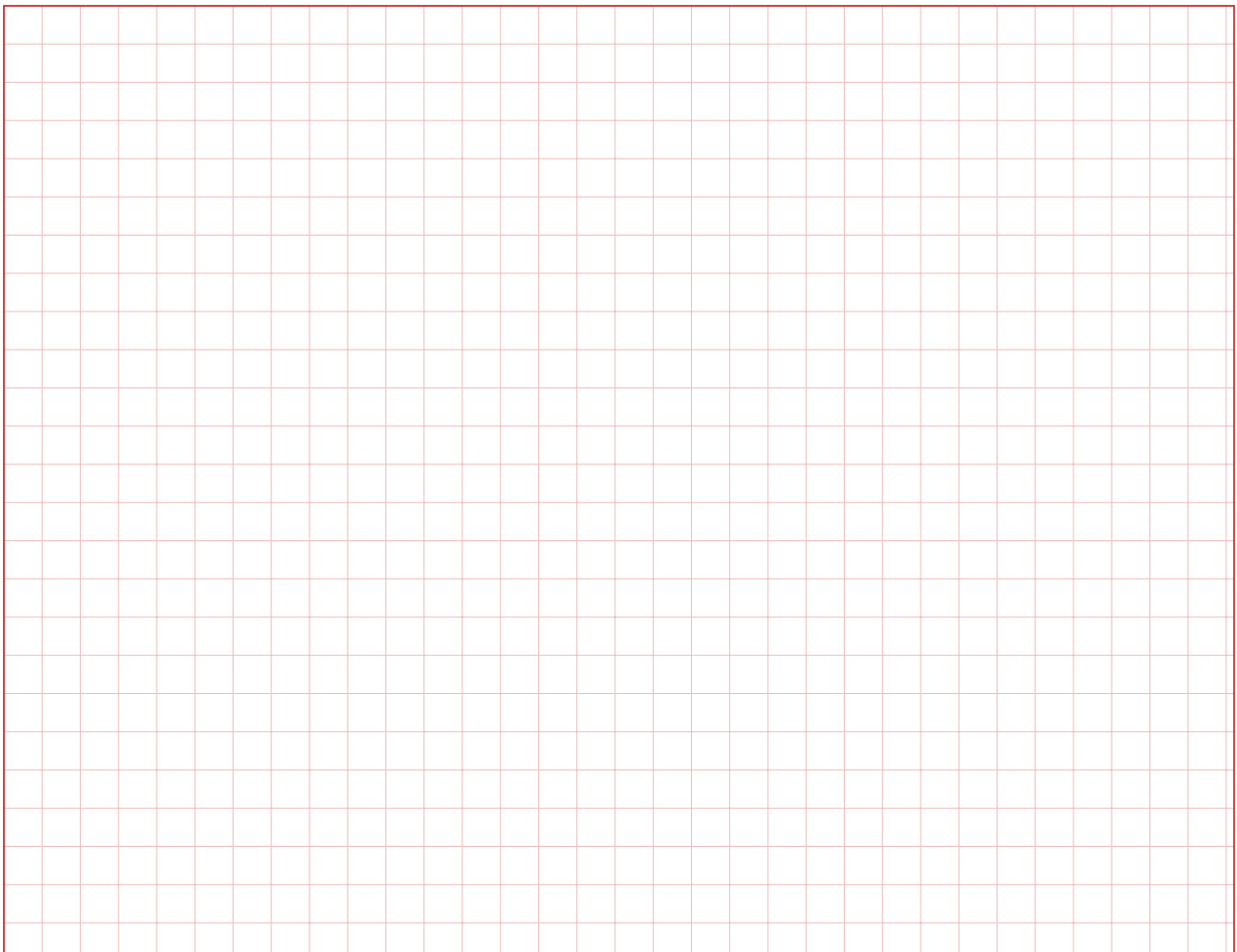
## Notizen



## Reelle Funktionen

- Einführung
- Die Ausnahmen zuerst
- Klasse RealFunction
- Kombination von Funktionen
- Differenzierbare Funktionen

## Notizen



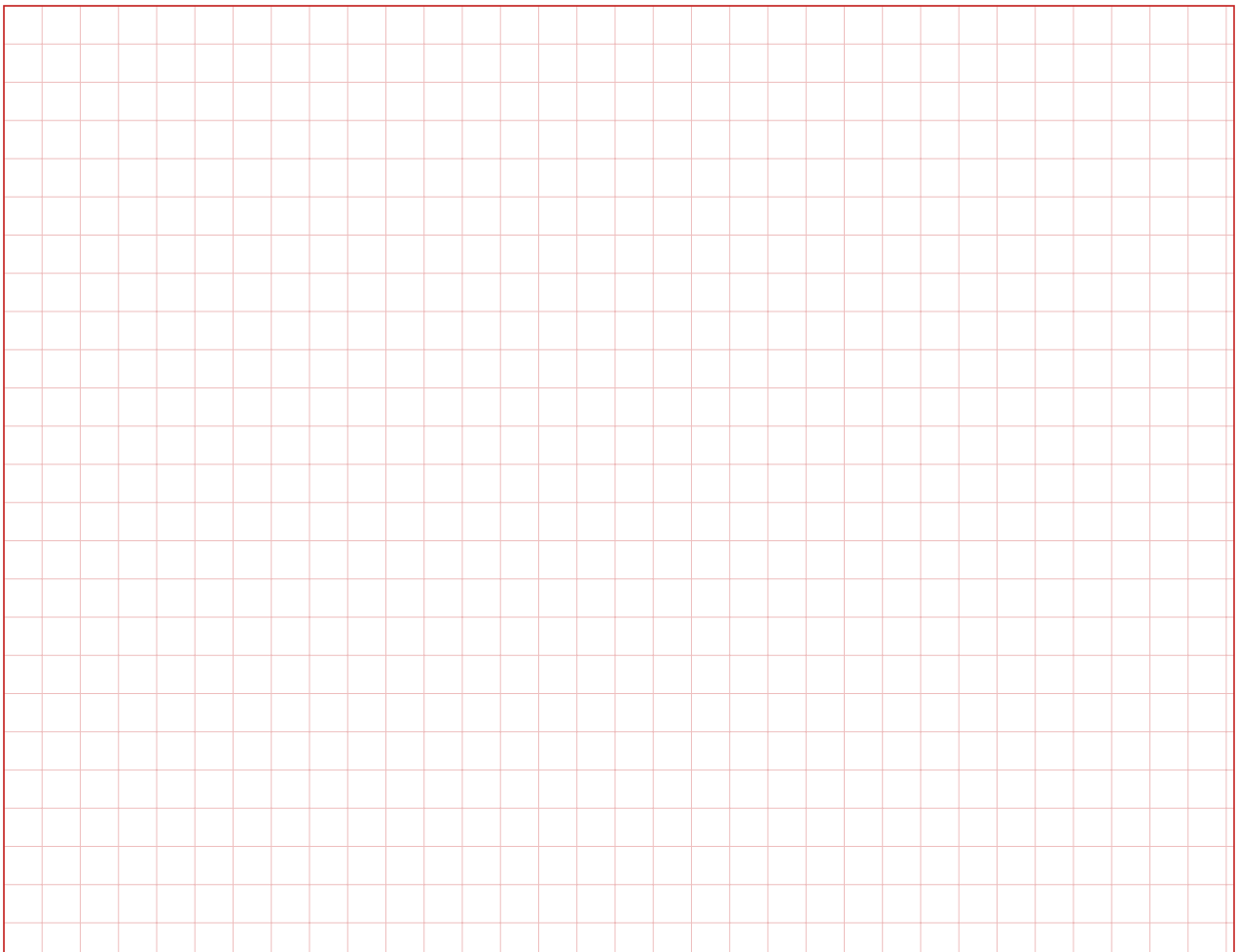
# Differenzierbare Funktionen

Wir wollen noch das **Ableiten** (Differenzieren) von Funktionen modellieren



- ✓ Implementieren Sie die **geprüfte Ausnahme** DiffException
  - ▶ Implementieren Sie die **drei Konstruktoren** wie in den Folien
  - ▶ Fügen Sie im Folgenden die Ausnahmen zu den **Methodensignaturen** hinzu wenn gefordert
- ✓ Implementieren Sie das **interface** Differentiable
  - ▶ Differentiable kann von **Funktionen mit einer Ableitung** implementiert werden
  - ▶ derive liefert die Ableitung als RealFunction und generiert eine DiffException im Fehlerfall

## Notizen

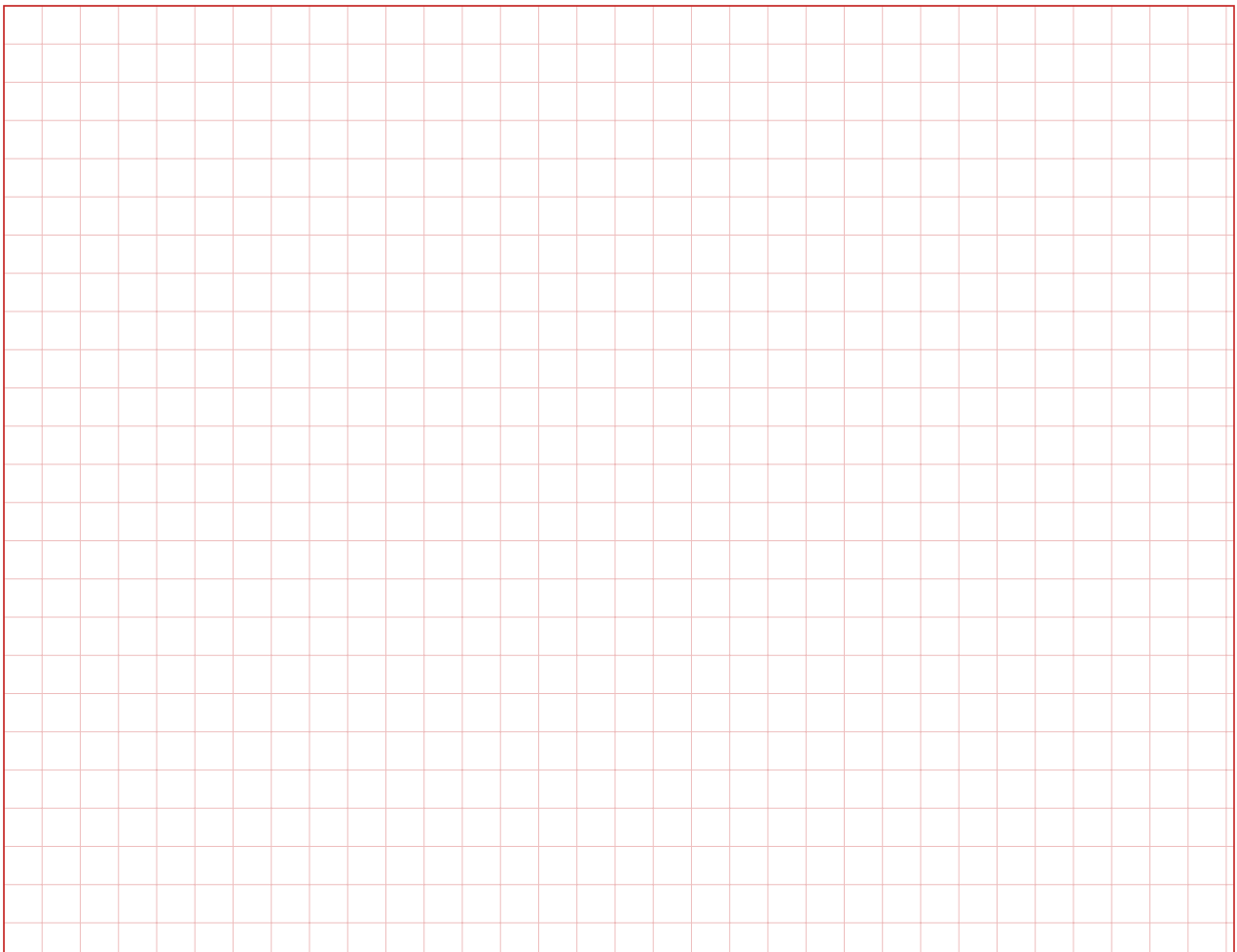




## Konkrete Differenzierbare Funktionen

- ✓ Implementieren Sie das Interface **Differentiable** für folgende Klassen
  - ▶ Constant → Constant mit  $c = 0$
  - ▶ Sine:  $a \cdot \sin(f \cdot x) \rightarrow$  Cosine mit  $a \cdot f \cdot \cos(f \cdot x)$
  - ▶ Cosine:  $a \cdot \cos(f \cdot x) \rightarrow$  Sine mit  $-a \cdot f \cdot \sin(f \cdot x)$
  - ▶ CubicPolynomial:  $a \cdot x^3 + b \cdot x^2 + c \cdot x + d \rightarrow$  CubicPolynomial mit  $0 \cdot x^3 + 3 \cdot a \cdot x^2 + 2 \cdot b \cdot x + c$
  - ▶ Exp: → Exp
  - ▶ Log siehe unten
- ▶ Verwenden Sie einen **kovarianten** Rückgabetypen für derive, z.B. kann Sine.derive den Rückgabetyp Cosine verwenden
- ✓ Für derive von Log gehen Sie wie folgt vor
  - ▶ Definieren Sie eine **private statische geschachtelte** Klasse in Log mit dem Namen LogDerivative
  - ▶ inDomain(x) liefert **true** wenn  $x > 0$
  - ▶ evaluateAt(x) ==  $1/x$
  - ▶ Geben Sie in Log.derive() eine Instanz von LogDerivative zurück
- ▶ Verwenden Sie zum Testen `calculus/DifferentiableFunctionTest.java`

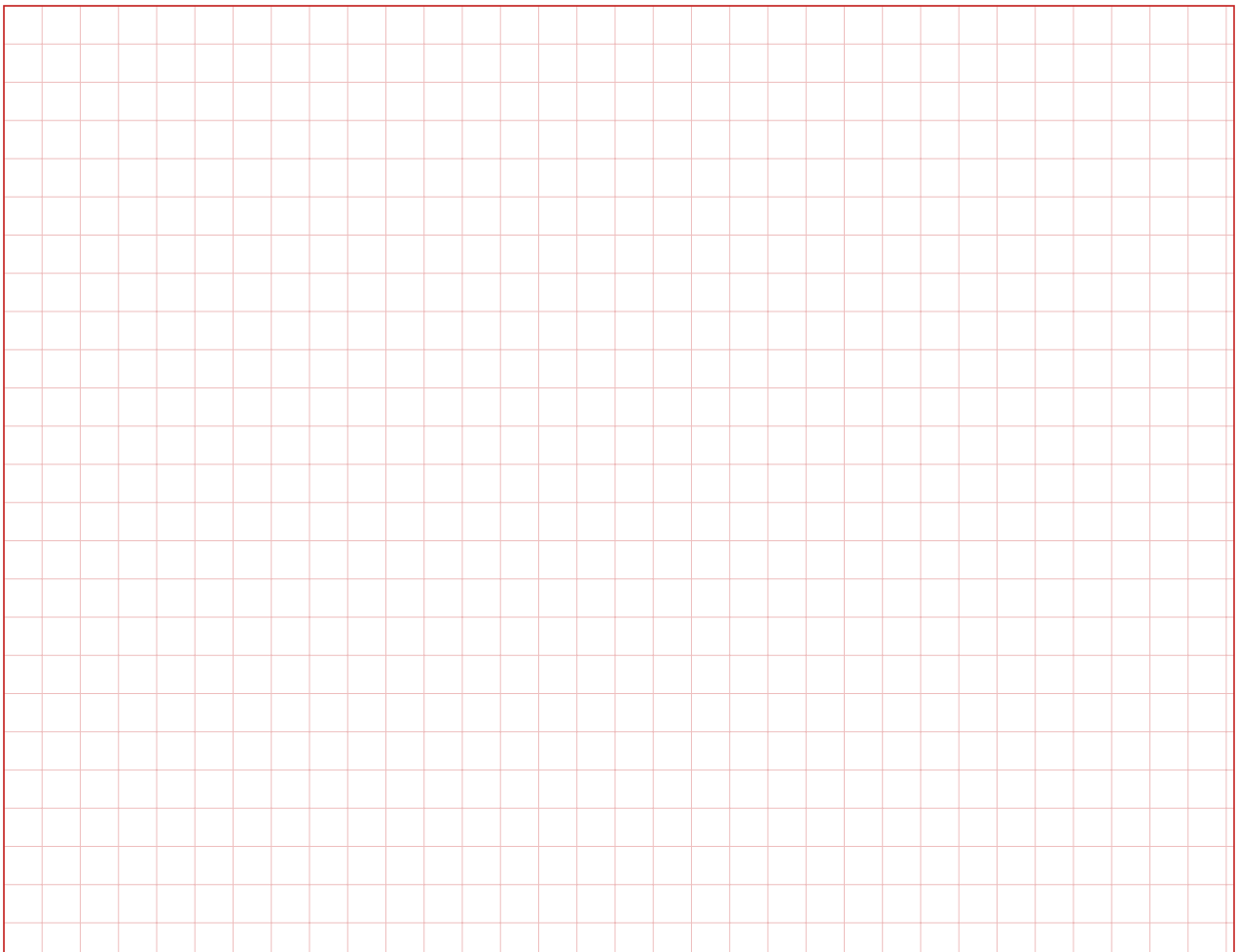
### Notizen



# Differenzieren der Operationen

- ▶ Sie haben es bald geschafft: Das ist die letzte Aufgabe!
- ▶ Implementieren Sie das Interface `Differentiable` für die binären Operationen
  - ▶ **Hinweis:** Es handelt sich jeweils um die **Ableitungsregeln** bekannt aus „Mathematik II“
  - ▶ Wenn einer der beiden Operanden `Differentiable` **nicht** implementiert, generieren Sie eine **`DiffException`**
  - ▶ Im folgenden bezeichnet `l` den linken und `r` den rechten Operanden und `l'` und `r'` deren Ableitungen
  - ▶ Addition → `new Addition(l', r')`
  - ▶ Multiplication → `new Addition( new ←  
Multiplication(l,r'), new Multiplication(l',r))`
  - ▶ Composition →  
`new Multiplication(new Composition(l',r), r')`
- ▶ Verwenden Sie zum Testen
  - 📄 `calculus/DifferentiableBinaryOperationTest.java` und/oder
  - 📄 `calculus/Plotter.java`

## Notizen



## Bonusaufgabe: Newton-Verfahren

- ▶ Sie haben also noch **nicht genug**?
- ▶ **Bonusaufgabe (optional)**: Implementieren Sie in der Klasse `RealFunction` das Newton-Verfahren
  - ▶ <https://de.wikipedia.org/wiki/Newtonverfahren>
  - ▶ Das **Newton-Verfahren** nähert eine **Nullstelle** einer Funktion mithilfe der Ableitung an
  - ▶ **Deklaration** in `RealFunction`:  
`public double newton(double start, double error)`
  - ▶ **Verfahren**
    - ▶ Starte mit  $x = \text{start}$
    - ▶ Solange `Math.abs(f(x)) > error` ist...
    - ▶ Berechne  $x = x - f(x)/f'(x)$  wobei  $f'(x)$  die Ableitung von  $f$  ist
    - ▶ Gib  $x$  zurück wenn `Math.abs(f(x)) <= error`
  - ▶ `(new Sine(1,1)).newton(3,1e-5)` sollte ungefähr  $\pi$  liefern

### Notizen

