

# Funktionale und objektorientierte Programmierkonzepte



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Präsenz-Sprechstunde B

Simon Hock, Nhan Huynh, Daniel Mangold



Statischer vs. dynamischer Typ

Statischer Typ  $\neq$  `static`

Racket HtDPL

Rekursion

Vergleich iterativ - rekursiv

Exkurs: Stack

Arbeitsphase



## ■ Statischer Typ:

- ▣ Typ, der bei der Variablendeklaration angegeben wird
- ▣ Ist bei der Kompilierung bekannt

## ■ Dynamischer Typ:

- ▣ Typ des tatsächlichen Objekts
- ▣ Ist erst zur Laufzeit bekannt
- ▣ Kann vom statischen Typ abweichen → Gleich oder Subtyp des statischen Typs

## ■ Formaler Aufbau

`<Statischer Typ> Bezeichner = <Dynamischer Typ>;`

## Klasse Rectangle

```
1 public class Rectangle {
2     private final double width;
3     private final double height;
4
5     public Rectangle(double width, double height) {
6         this.width = width;
7         this.height = height;
8     }
9
10    public double getWidth() { return width; }
11    public double getHeight() { return height; }
12    public double getArea() { return width * height; }
13    public double getCircumference() { return 2 * (width + height); }
14 }
```

## Klasse Square

```
1 public class Square extends Rectangle {
2     public Square(double length) {
3         super(length, length);
4     }
5
6     @Override
7     public double getCircumference() {
8         return 4 * getWidth();
9     }
10
11     public double getDiagonalLength() {
12         return Math.sqrt(2 * getWidth() * getWidth());
13     }
14 }
```

- Auf welche Methoden können wir mit `rectangle` zugreifen?
- Auf welche Methoden können wir mit `square` zugreifen?
- Auf welche Methoden können wir mit `rs` zugreifen?

## Vererbung Klasse Rectangle und Square

```
1 Rectangle rectangle = new Rectangle(22,4);  
2 Square square = new Square(5);  
3 Rectangle rs = new Square(21);
```

---

**Information:** Ein zentraler Punkt bei Subtypen ist also, dass man beim dynamischen Typ statt der eigentlichen Klassen ebenfalls ihre Subtypen verwenden kann. Das gilt für die Deklaration von Attributen, Variablen, sowie dem Parameterwert und Rückgabetyt von Methoden.

Beim Parameterwert wird deshalb zwischen dem aktuellen Wert (der Parameterwert, der eingesetzt wird) und dem formalen Wert (der in der Definition des Parameters in der Methode steht) unterschieden.

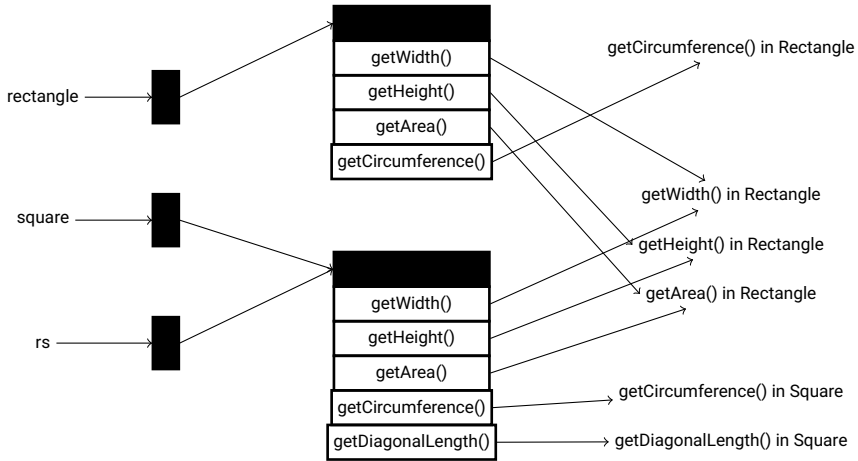


Abbildung: Methodentabelle bezüglich Rectangle und Square Beispiel



- Statischer Typ: Typ einer Variable, die bei der Variablendeklaration angegeben ist.
- Schlüsselwort `static`: Klassenattribute/ -methoden (statisch), die nicht einzelnen Objekten, sondern deren gesamter Klasse zugeordnet werden.

## Schlüsselwort `static`

```
1 public static int sum(int n) {  
2     int result = 0;  
3     for (int k = 0; k <= n; k++) {  
4         result += k;  
5     }  
6     return result;  
7 }
```

- Sprachumfang: How to Design a Program Language
- Funktionale Sprache
- Einschränkung auf wenige Konstrukte und vor allem Rekursion
- Atomare Ausdrücke
- Syntax: Präfix-Notation

Notation	Addition zweier Zahlen
Präfix	$+ x y$
Infix	$x + y$
Postfix	$x y +$

**Tabelle:** Überblick von Notationen

- Schlüsselwort: `define`

- Formaler Aufbau:

`(define (Bezeichner Argumente*)Körper)`

- Bezeichner: Name der Funktion
  - Argumente: Optional, werden mit einem Leerzeichen getrennt
  - Körper: Atomarer Ausdruck
- Immer wenn einer Klammer geöffnet wird, wird ein Funktionsaufruf erwartet!

# Vergleich Java und Racket



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Methode add

```
1 public static int add(int x, int y) {  
2     return x + y;  
3 }
```

## Funktion add

```
1 (define (add x y) (+ x y))
```



„Eine Rekursion ist eine Funktion, die sich selbst aufruft. Dabei wird versucht das Problem auf eine einfachere Variante des **gleichen** Problems rekursiv zu reduzieren, welches dann gelöst wird.“

- **Rekursionsanker:** Ab diesem Punkt bricht die Rekursion ab. (Abbruchbedingung)
- In Zeile 4 ruft die Funktion sich selbst auf.
  - ▣ Ohne Rekursionsanker würde die Funktion sich selbst *endlos* aufrufen. (Endlosrekursion)

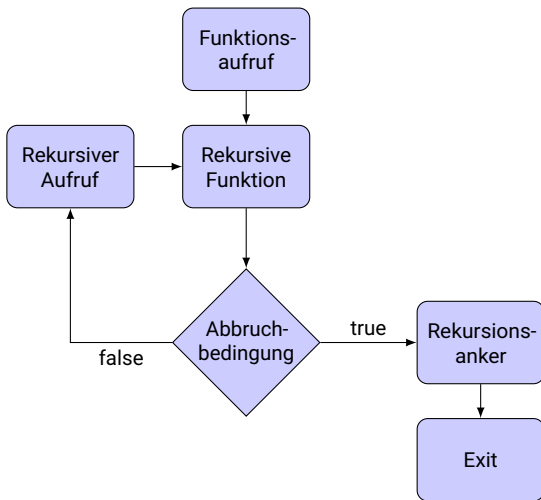
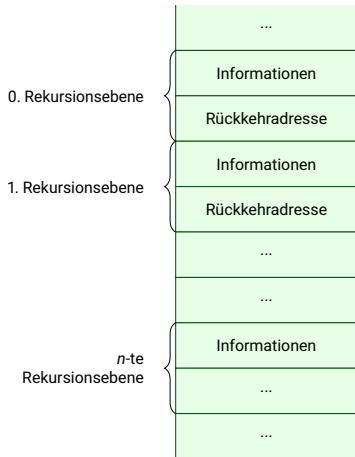


Abbildung: Flow-Chart zu rekursiven Aufruf



**Abbildung:** Abstrakte Visualisierung eines Stacks von rekursiven Funktionen



Wir möchten die Gaußsche Summenformel implementieren.

$$0 + 1 + 2 + \dots + n = \sum_{k=0}^n k = k \quad (1)$$





## Gaußsche Summenformel

```
1 public static int sum(int n) {  
2     int result = 0;  
3     for (int k = 0; k <= n; k++) {  
4         result += k;  
5     }  
6     return result;  
7 }
```



## Gaußsche Summenformel

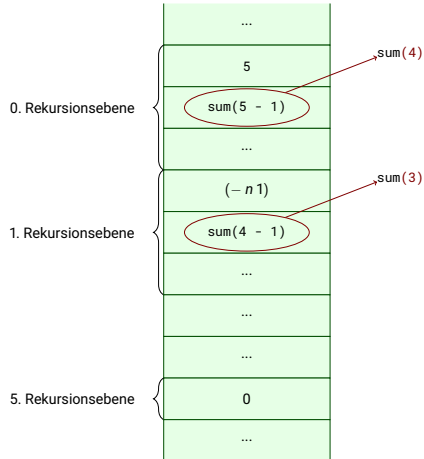
```
1 public static int sum(int n) {  
2     if (n == 0) {  
3         return 0;  
4     }  
5     return sum(n - 1) + n;  
6 }
```

# Gaußsche Summenformel - Rekursiv Visualisierung

Rekursionsebene	Aktueller Aufruf	Zwischenstand
0	$\text{sum}(5)$	$\text{sum}(4) + 5$
1	$\text{sum}(4)$	$\text{sum}(3) + 4 + 5$
2	$\text{sum}(3)$	$\text{sum}(2) + 3 + 4 + 5$
3	$\text{sum}(2)$	$\text{sum}(1) + 2 + 3 + 4 + 5$
4	$\text{sum}(1)$	$\text{sum}(0) + 1 + 2 + 3 + 4 + 5$
5	Zurück zum Aufruf von $\text{sum}(0)$	$0 + 1 + 2 + 3 + 4 + 5$
4	Zurück zum Aufruf von $\text{sum}(1)$	$1 + 2 + 3 + 4 + 5$
3	Zurück zum Aufruf von $\text{sum}(2)$	$3 + 3 + 4 + 5$
2	Zurück zum Aufruf von $\text{sum}(3)$	$6 + 4 + 5$
1	Zurück zum Aufruf von $\text{sum}(4)$	$10 + 5$
0	Zurück zum Aufruf von $\text{sum}(5)$	15



- Jede Rekursionsebene benötigt Informationen für die Berechnung
- Benötigte Informationen von jeder Rekursionsebene werden im Normalfall in den Stack abgelegt.



**Abbildung:** Abstrakte Visualisierung des Stacks von rekursiven Aufruf von `sum`

## Selbstständiges Arbeiten