Funktionale und objektorientierte Programmierkonzepte

TECHNISCHE UNIVERSITÄT DARMSTADT

Präsenz-Sprechstunde B

Simon Hock, Nhan Huynh, Daniel Mangold

Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

Statischer vs. dynamischer Typ
- Statischer Typ $\neq$ `static`
- Beispiel - Athene Karte

Racket HtDPL

Rekursion
- Vergleich iterativ - rekursiv
- Exkurs: Stack

Arbeitsphase

Statischer vs. dynamsicher Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

- **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 $\rightarrow$ Gleich oder Subtyp des statischen Typs
- Formaler Aufbau

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

Klasse Rectangle

```java
public class Rectangle {
  private final double width;
  private final double height;

  public Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }

  public double getWidth() { return width; }
  public double getHeight() { return height; }
  public double getArea() { return width * height; }
  public double getCircumference() { return 2 * (width + height); }
}
```

Klasse Square

```java
public class Square extends Rectangle {
  public Square(double length) {
    super(length, length);
  }

  @Override
  public double getCircumference() {
    return 4 * getWidth();
  }

  public double getDiagonalLength() {
    return Math.sqrt(2 * getWidth() * getWidth());
  }
}
```

- 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 Rückgabetyp von Methoden.

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

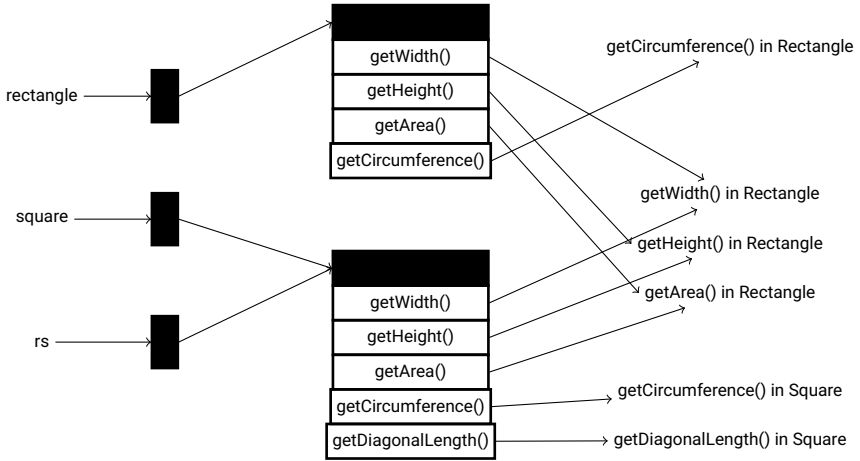

Abbildung: Methodentabelle bezüglich Rectangle und Square Beispiel

Statischer Typ $\neq$ `static`

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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 }
```

Beispiel - Athene Karte

TECHNISCHE UNIVERSITÄT DARMSTADT

- Modellierung der Athene Karte
- Enthält Namen eines Universitätsmitglied und weitere Information

Abbildung: Quelle: https://www.physik.tu-darmstadt.de/internationales/incoming/general_information_1/athene_card_1/athenecard.en.jsp

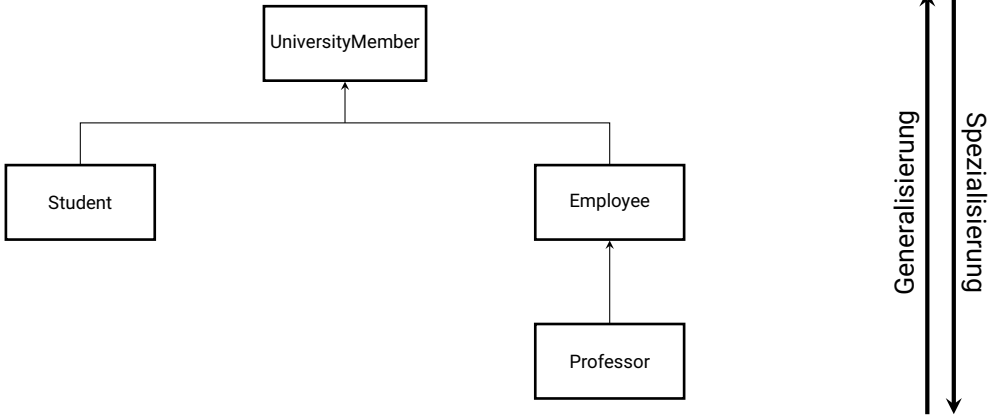

Abbildung: Typhierarchie - Universitätsmitglieder

Klasse UniversityMember

```
1  public abstract class UniversityMember {
2
3    protected final long id;
4    private String firstName;
5    private String lastName;
6
7    public UniversityMember(String firstName, String lastName, long id) {
8      this.id = id;
9      this.firstName = firstName;
10     this.lastName = lastName;
11   }
12
13   public abstract String getID();
14
```

```
15    public String getFirstName() {
16      return firstName;
17    }
18    public void setFirstName(String firstName) {
19      this.firstName = firstName;
20    }
21
22    public String getLastName() {
23      return lastName;
24    }
25    public void setLastName(String lastName) {
26      this.lastName = lastName;
27    }
28  }
```

Dokumentation von nicht implementierten Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

- Spezifiziert die Funktionalität der Methode, welche von Subklassen eingehalten werden müssen
- Subklassen können den Vertrag spezifizieren

```java
/**
 * Returns the unique identification of this university member.
 *
 * @return the unique identification of this university member
 */
public abstract String getId();
```

Klasse Student

```
1  public class Student extends UniversityMember {
2
3    private static long ID = 0;
4    private int semester;
5
6    public Student(String firstName, String lastName) {
7      super(firstName, lastName, ID++);
8      this.semester = 1;
9    }
10
11   @Override
12   public String getID() {
13     return String.format("Enrolment number: %07d", id);
14   }
```

```
15
16    public int getSemester() {
17      return semester;
18    }
19
20    public void setSemester(final int semester) {
21      this.semester = semester;
22    }
23  }
```

```java
/**
 * Returns the unique identification of this student. The identification
 * represents the enrollment number of this student.
 *
 * @return the unique identification of this student
 */
@Override
public String getID() {
  return String.format("Enrolment number: %07d", id);
}
```

Enum Department

```
1 public enum Department {
2   BIOLOGY,
3   CHEMISTRY,
4   COMPUTER_SCIENCE,
5   MATHEMATICS,
6   PHYSICS,
7 }
```

Klasse Employee

```java
public class Employee extends UniversityMember {
  private static long ID = 0;

  private Department department;

  public Employee(String firstName, String lastName, Department department) {
    super(firstName, lastName, ID++);
    this.department = department;
  }

  @Override
  public String getID() {
    return String.format("Person number: %07d", id);
  }
```

```
15
16    public Department getDepartment() {
17      return department;
18    }
19
20    public void setDepartment(final Department department) {
21      this.department = department;
22    }
23  }
```

Klasse Professor

```
1  public class Professor extends Employee{
2
3    private String room;
4
5    public Professor(String firstName, String lastName,
6                     Department department, String room) {
7      super(firstName, lastName, department);
8      this.room=room;
9    }
10
11   public String getRoom() {
12     return room;
13   }
14
```

```
15   public void setRoom(final String room) {
16     this.room = room;
17   }
18 }
```

Klasse AthenaCard

```
1  public class AthenaCard {
2    private static long ID = 0;
3
4    private final UniversityMember member;
5    private final long cardID;
6
7    public AthenaCard(final UniversityMember member) {
8      this.member = member;
9      this.cardID = ID++;
10   }
11
12   public UniversityMember getMember() { return member; }
13
14   public String getCardID() { return String.format("%17d", id); }
```

```
15
16    public String getFirstName() { return member.getFirstName(); }
17
18    public String getLastName() { return member.getLastName(); }
19
20    public String getID() { return member.getId(); }
21 }
```

Racket HtDPL

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

Funktionen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

```java
public static int add(int x, int y) {
  return x + y;
}
```

Funktion add

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

Rekursion

TECHNISCHE UNIVERSITÄT DARMSTADT

„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)

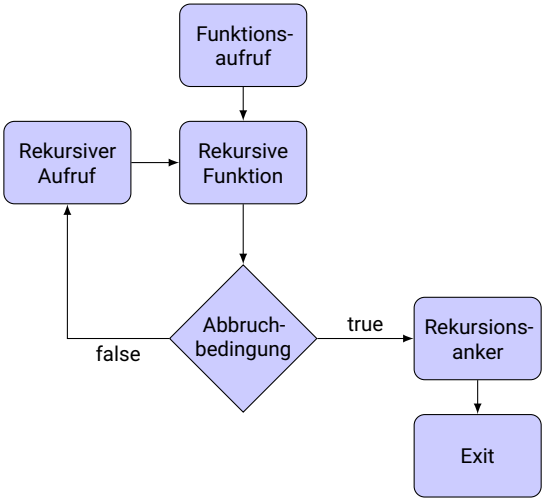

Abbildung: Flow-Chart zu rekursiven Aufruf

Vergleich iterativ - rekursiv

TECHNISCHE UNIVERSITÄT DARMSTADT

Wir möchten die Gaußsche Summenformel implementieren.

$$0 + 1 + 2 + ... + n = \sum_{k=0}^{n} = k \tag{1}$$

Gaußsche Summenformel - Iterativ

TECHNISCHE UNIVERSITÄT DARMSTADT

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 - Rekursiv

TECHNISCHE UNIVERSITÄT DARMSTADT

Gaußsche Summenformel

```java
public static int sum(int n) {
  if (n == 0) {
    return 0;
  }
  return sum(n - 1) + n;
}
```

Gaußsche Summenformel - Rekursiv Visualisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursionsebene	Aktueller Aufruf	Zwischenstand
0	`sum(5)`	`sum(4)+ 5`
1	`sum(4)`	`sum(3)+ 4 + 5`
2	`sum(3)`	`sum(2)+ 3 + 4 + 5`
3	`sum(2)`	`sum(1)+ 2 + 3 + 4 + 5`
4	`sum(1)`	`sum(0)+ 1 + 2 + 3 + 4 + 5`
5	Zurück zum Aufruf von `sum(0)`	`0 + 1 + 2 + 3 + 4 + 5`
4	Zurück zum Aufruf von `sum(1)`	`1 + 2 + 3 + 4 + 5`
3	Zurück zum Aufruf von `sum(2)`	`3 + 3 + 4 + 5`
2	Zurück zum Aufruf von `sum(3)`	`6 + 4 + 5`
1	Zurück zum Aufruf von `sum(4)`	`10 + 5`
0	Zurück zum Aufruf von `sum(5)`	`15`

Exkurs: Stack

TECHNISCHE UNIVERSITÄT DARMSTADT

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

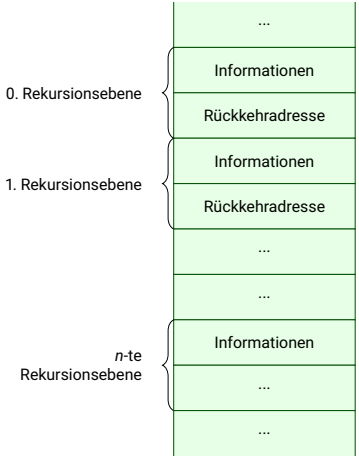

Abbildung: Abstrakte Visualisierung eines Stacks von rekursiven Funktionen

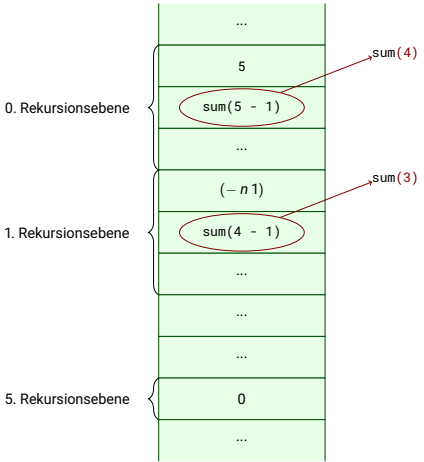

Abbildung: Abstrakte Visualisierung des Stacks von rekursiven Aufruf von `sum`

Selbstständiges Arbeiten