TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

Subtyping revisited

Subtyping und Vererbung I

Wir betrachten im Folgenden nochmals den Zusammenhang zwischen Subtypbildung und Vererbung.

Für Java ist das insbesondere der Zusammenhang von

- Schnittstellentypen
- Klassentypen

Die Deklaration eines Typs T legt dabei die direkten Supertypen von T fest.

Subtypbildung in Java: Klassen I

Syntax der Klassendeklaration:

```
<Modifikatorenliste> class <Klassenname>
     [  extends <Klassenname> ]
     [  implements <Liste von Schnittstellennamen> ]
{
    <Liste von Attribut-,
     Konstruktor-,
     Methodendeklaration>
}
```

Eine Klassen **erweitert** eine direkte Superklasse und **implementiert** ggf. mehrere Schnittstellentypen.

Subtypbildung in Java: Klassen II

- Gibt es keine `extends`-Klausel, ist `Object` die direkte Superklasse.

  Andernfalls ist die in der `extends`-Klausel genannte Klasse die direkte Superklasse.

- Die in der `implements`-Klausel genannten Schnittstellentypen müssen implementiert werden.

- Die Superklasse und alle implementierten Schnittstellentypen sind Supertypen der deklarierten Klasse.

Subtypbildung in Java: Interfaces

Syntax der Interface-Deklaration:

```
<Modifikatorenliste> interface <Schnittstellenname>
    [  extends <Liste von Schnittstellennamen> ]
{
   <Liste von Konstanten und Methodensignaturen>
}
```

Bei einer Schnittstellendeklaration gilt Folgendes:

- Gibt es keine `extends`-Klausel, ist `Object` der einzige Supertyp.
- Andernfalls sind die in der `extends`-Klausel genannten Schnittstellentypen die direkten Supertypen.

Substituierbarkeit

Prinzip der Substituierbarkeit (Liskovsches Substitutionsprinzip):

Sei $S \leq T$; dann ist an allen Programmstellen, an denen ein Objekt vom Typ T zulässig ist, auch ein Objekt vom Typ S zulässig.

Konsequenzen:

$\Rightarrow$ Subtypobjekte müssen alle Eigenschaften des Supertyps aufweisen.

Realisierung von Klassifikationen

Die Klassen bzw. Begriffe in einer Klassifikation können im Programm durch Schnittstellen- oder Klassentypen realisiert werden.

Wir betrachten die Klassifikation aus Abschnitt 13 bestehend aus:

Person, Printable, Student, Angestellte,
WissAngestellte und VerwAngestellte,
RegulaererStudent und AustauschStudent.

Variante 1

Nur die Blätter der Klassifikation (`RegulaererStudent`, `AustauschStudent`, `WissAngestellte`,`VerwAngestellte`) werden durch Klassen realisert, alle anderen durch Schnittstellen.

```
interface Printable {
  void print();
}

interface Person {
  String getName();
  String getBirthdate();
}

interface Angestellte extends Person, Printable { ... }
interface Student     extends Person, Printable { ... }

class WissAngestellte   implements Angestellte { ... }
class VerwAngestellte   implements Angestellte { ... }
class RegulaererStudent implements Student   { ... }
class AustauschStudent  implements Student   { ... }
```

Aufgabe: Visualisieren Sie die entsprechende Typhierarchie!

2. Variante

Bis auf den Typ `Printable` realisieren wir alle Typen durch Klassen:

```
interface Printable { ... }

class Person        { ... }

class Student     extends Person implements Printable { ... }
class Angestellte extends Person implements Printable { ... }

class WissAngestellte   extends Angestellte { ... }
class VerwAngestellte   extends Angestellte { ... }

class RegulaererStudent extends Student   { ... }
class AustauschStudent  extends Student   { ... }
```

Klassendiagramm zur 2. Variante

Diskussion

Verwendung von Schnittstellen in Java:

- nur wenig über den Typ bekannt
- keine Festlegung von Implementierungsteilen
- als Supertyp von Klassen mit mehreren Supertypen

Verwendung von Klassen in Java:

- Objekte sollen von dem Typ erzeugt werden
- Vererbung an Subtypen soll ermöglicht werden

Das Quadrat-Rechteck-Problem

Die `Rectangle`- und `Square`-Klasse

```
class Rectangle extends AFigure {
  private double length;
  private double width;
  ...
  double area() {
    return length * width;
  }
}

class Square extends AFigure {
  private double length;
  ...
  double area() {
    return length * length;
  }
}
```

Das Quadrat-Rechteck-Problem

Wie sollte die Vererbungs-/Typrelation zwischen diesen beiden Klassen aussehen?

Variante 1: `Square` als Superklasse von `Rectangle` I

```
class Square extends AFigure {
  private double length;
  ...
  double area() {
    return length * length;
  }
}
class Rectangle extends Square {
  private double width;
  ...
  double area() {
    return length * width;
  }
}
```

Variante 1: `Square` als Superklasse von `Rectangle` II

- Vererbung des `length`-Attributs möglich
- Methoden müssen überschrieben werden
- Ein Rechteck ist aber im allgemeinen kein Quadrat...

Variante 2: `Rectangle` als Superklasse von `Square` I

```
class Rectangle extends AFigure {
  private double length;
  private double width;
  Rectangle (CartPt loc, double length, double width) {
    super(loc);
    this.length = length;
    this.width = width;
  }
  double area() {
    return length * width;
  }
}
class Square extends Rectangle {
  Square (CartPt loc, double length) {
    super(loc, length, length);
  }
}
```

Variante 2: `Rectangle` als Superklasse von `Square` II

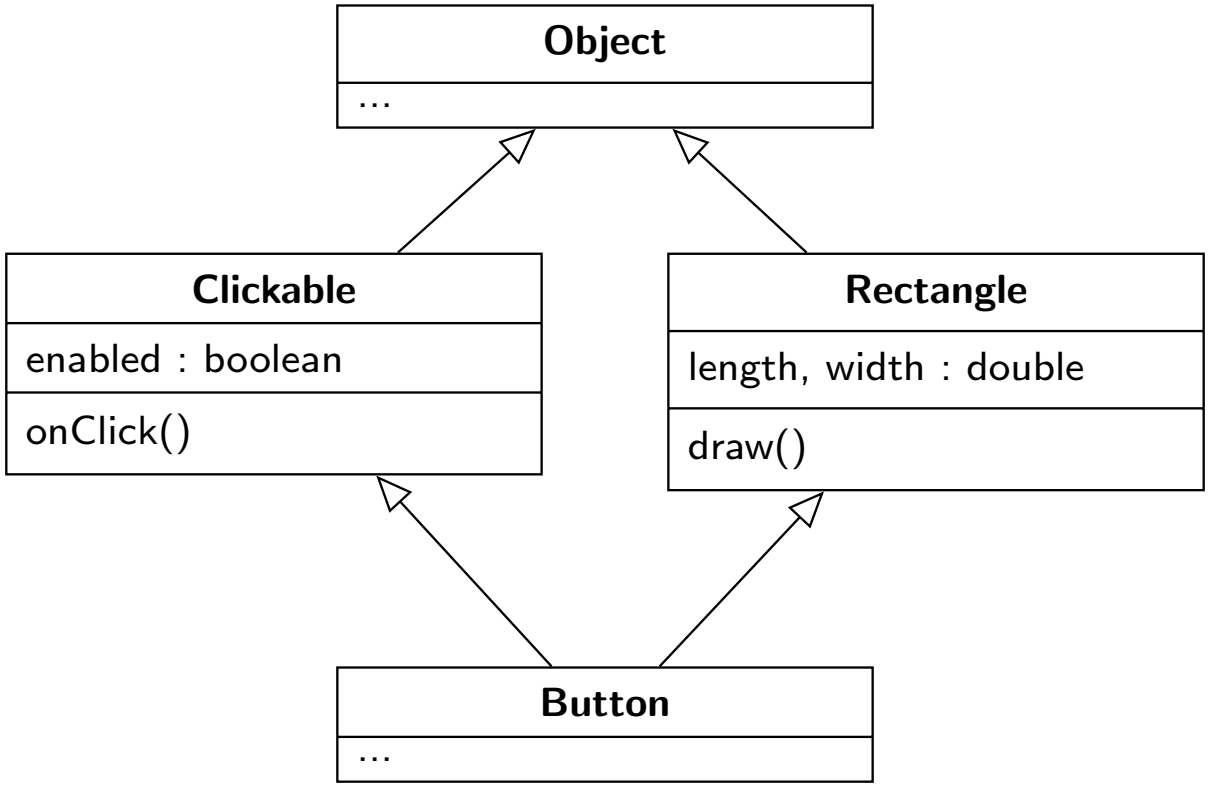
- Methoden müssen nicht überschrieben werden, aber spezialisierte Methoden könnten effizienter sein
- `Square`-Objekte haben nicht verwendbare, unnötige Attribute
- Invariante `length == width` muss für `Square`-Objekte immer sichergestellt sein!

Weitere Varianten

- Verzicht auf Vererbungsbeziehung
  - Code-Duplikation
  - Keine Subtyp-Polymorphie anwendbar
- Einführung einer zusätzlichen gemeinsamen Superklasse, die die Gemeinsamkeiten abstrahiert
  - Unnötiges Aufblähen der Klassenhierarchie
  - Keine Subtyp-Polymorphie anwendbar zwischen Quadrat und Rechteck

Mehrfachvererbung I

Übernimmt eine Klasse Programmteile von mehreren anderen Klassen spricht man von **Mehrfachvererbung** (engl. *multiple inheritance*).



Mehrfachvererbung II

Da in Java (bis Version 1.7) Mehrfachvererbung nicht möglich ist, muss man die Modellierung durch Einfachvererbung und mehrfache Subtypbildung umsetzen.

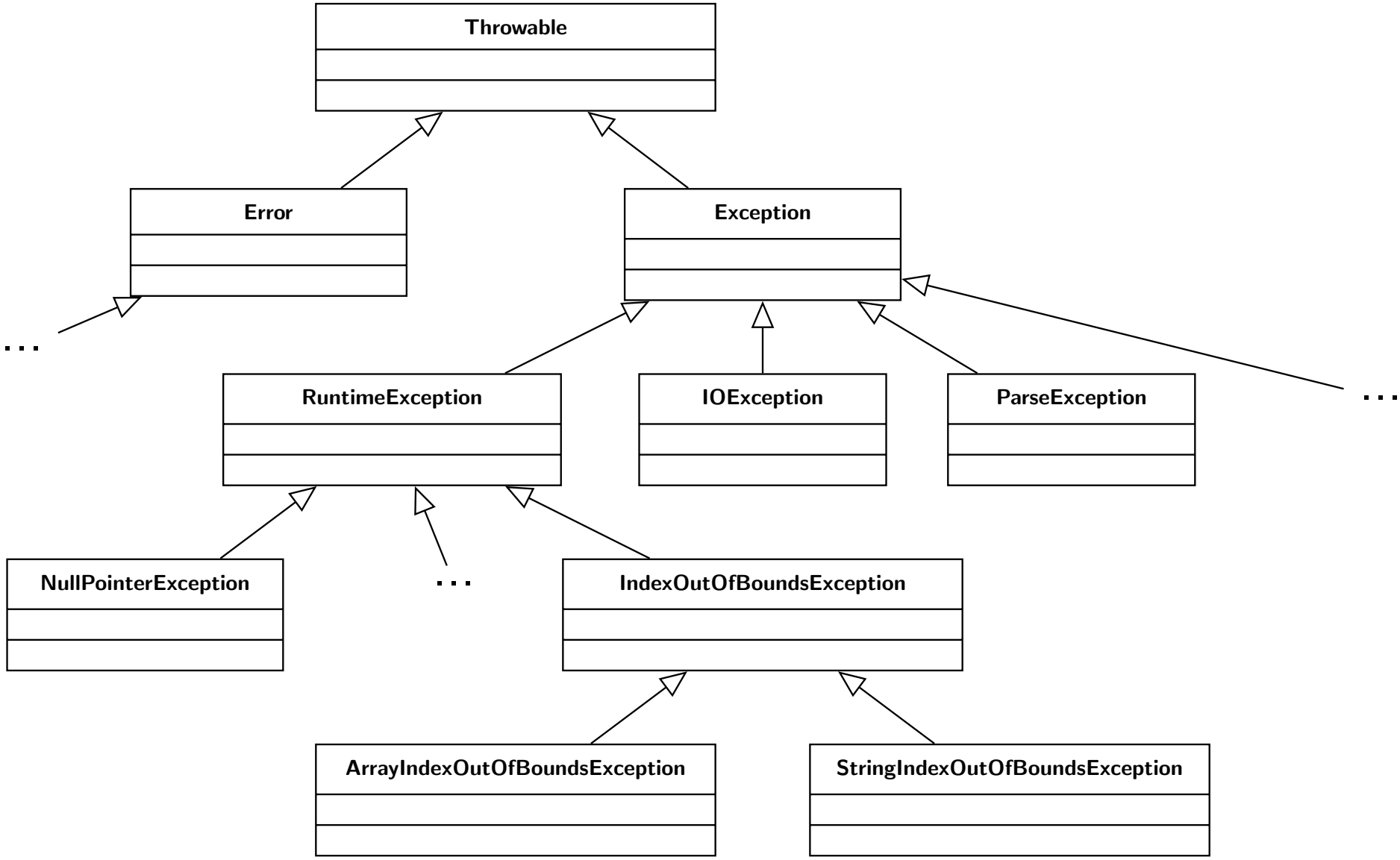
Seit Java 8 können Interfaces default-Implementierungen von Methoden enthalten. Dadurch kann eine Art von Mehrfachvererbung umgesetzt werden.

Exceptions

Objekt-orientierte Ausnahmebehandlung in Java

- Ausnahmen werden in Java durch Objekte repräsentiert; d.h. wenn eine Ausnahme auftritt, wird ein entsprechendes Ausnahmeobjekt erzeugt.
- Ausnahmen können entweder sprachdefiniert sein oder durch eine `throw`-Anweisung ausgelöst werden.
- Die `throw`-Anweisung verlangt eine Referenz auf ein Objekt vom Typ `Throwable`.

Die Klassenhierarchie der Exceptions (Ausschnitt)



Klassifizierung von Ausnahmen und Fehlern

- Die Klasse `Throwable` aus dem Packet `java.lang` ist die Superklasse aller Ausnahmen- und Fehlerklassen in Java.
- `Error`s sind schwerwiegende Fehler, die in der Regel nicht vom Programm abgefangen und behandelt werden sollten.
- `Exception`s sollten vom Programm abgefangen und geeignet behandelt werden.

Beispiel: Ausnahmebehandlung

```
1  public class UeberlaufException extends RuntimeException {}
2
3  public class ExceptionTest {
4   public static void main(String[] args) {
5    try {
6     int ergebnis = 0;
7     int m = Integer.parseInt(args[0]);
8     int n = Integer.parseInt(args[1]);
9     long aux = (long) m + (long) n;
10    if (aux > Integer.MAX_VALUE) {
11       throw new UeberlaufException(); // selbstdefiniert
12    }
13    ergebnis = (int) aux;
14   } catch (IndexOutOfBoundsException e) {
15       System.out.println("Zuwenig Argumente");
16   } catch (NumberFormatException e) {
17       System.out.println("Parameter ist keine int-Konstante");
18   } catch (UeberlaufException e) {
19       System.out.println("Ueberlauf bei Addition");
20   }
21  }
22 }
```

Checked und Unchecked Exceptions

- In Java zeigt die `throws`-Klausel in einer Methodensignatur an, dass eine Methode Exceptions auslöst bzw. weitergibt, ohne sie selbst abzufangen.
- **Checked Exceptions** müssen, wenn sie in einer Methode geworfen werden, entweder dort in einem Exception Handler behandelt werden, oder aber in einer `throws`-Klausel in der Methodensignatur angegeben werden. Dies wird vom Compiler überprüft.
- **Unchecked Exceptions** müssen nicht entsprechend deklariert werden. Nur Exceptions vom Typ `RuntimeException` und `Error` sind unchecked.

Beispiel: Checked Exceptions

```
3  public static void main(String[] args) throws Exception {
4      double[] a = {};
5      System.out.println("avg = " + average(a));
6  }
7  static double average(double[] a) throws Exception {
8      checkPreconditions(a);
9      int sum = 0;
10     for (int i = 0; i < a.length; i++) {
11         sum += a[i];
12     }
13     return sum / a.length;
14 }
15 static void checkPreconditions(double[] a) throws Exception {
16     if (a == null) {
17         throw new Exception("Array darf nicht null sein");
18     }
19     if (a.length == 0) {
20         throw new Exception("Array darf nicht leer sein");
21     }
22 }
```

Reihenfolge der Exception-Handler

- Ein Handler für Exceptions einer Klasse `X` behandelt auch Exceptions aller von `X` abgeleiteten Klassen (Stichwort: Subtyp-Polymorphie).
- Die Reihenfolge der `catch`-Blöcke ist daher relevant bei der Ausnahmebehandlung.
- Zunächst müssen die Handler für die am meisten spezialisierten Klassen aufgelistet werden und dann mit zunehmender Generalisierung die Handler für die entsprechenden Superklassen. Dies wird vom Compiler überprüft.

Beispiel: Reihenfolge der Exception-Handler I

```
3  public class MyException extends Exception {
4    MyException() {
5      super("Ueberlauf bei der Integer-Addition");
6    }
7  }
```

- Der `Exception`-Konstruktor kann einen String-Parameter nehmen, der eine Beschreibung des Fehlers enthält.
- Die Fehlerbeschreibung kann durch die Methode `getMessage()` in der Fehlerbehandlung ausgelesen werden.

Beispiel: Reihenfolge der Exception-Handler II

```
3   public static void main(String[] args) {
4    try {
5     int ergebnis = 0;
6     int m = Integer.parseInt(args[0]);
7     int n = Integer.parseInt(args[1]);
8     long aux = (long) m + (long) n;
9     if (aux > Integer.MAX_VALUE) {
10       throw new MyException(); // selbstdefiniert
11    }
12    ergebnis = (int) aux;
13   } catch (ArrayIndexOutOfBoundsException e) {
14       System.out.println("Zuwenig Argumente");
15   } catch (Exception e) {
16       System.out.println("Ein Fehler ist aufgetreten");
17       e.getMessage();
18   }
19
20  }
```

Klassenattribute und -methoden

Klassenattribute

Die Deklaration eines **Klassenattributs** liefert eine klassenlokale Variable. Klassenattribute/-variablen werden häufig auch als **statische Attribute/Variablen** bezeichnet.

Syntax:

```
static <Typausdruck> <Attributname> ;
```

Die Variable kann innerhalb der Klasse mit dem Attributnamen, außerhalb mittels

```
<Klassenname> . <Attributname>
```

angesprochen werden.

Lebensdauer von Klassenattributen

Die Lebensdauer der Variablen entspricht der Lebensdauer der Klasse.
Sie werden beim Laden der Klasse initialisiert.

Bemerkung:
Klassenvariablen verhalten sich ähnlich wie globale Variablen in der prozeduralen Programmierung.

Beispiel: Klassenvariablen

```
class Uebungsgruppe {
  static int maxTeilnehmer = 30;
  private String name;
  private int teilnehmer;

  Uebungsgruppe(String name){
    this.name = name;
    this.teilnehmer = 0;
  }  ...

}

...
public static void main (String[] args) {
    Uebungsgruppe u = new Uebungsgruppe("Montagsgruppe");

    String s = u.getName();
    int i = Uebungsgruppe.maxTeilnehmer;
}
```

Klassenmethode

Die Deklaration einer **Klassenmethode** entspricht der Deklaration einer Prozedur. Sie werden häufig auch als **statische Methoden** bezeichnet.

Klassenmethoden besitzen keinen impliziten Parameter.

Sie können nur auf Klassenattribute, Parameter und lokale Variable zugreifen.

Syntax:

```
static <Methodendeklaration>
```

Klassenmethoden werden mit folgender Syntax aufgerufen:

```
<Klassenname> . <Methodenname> ( ... )
```

Innerhalb der Klasse kann der Klassenname entfallen.

Beispiel: Klassen-, statische Methoden

Deklaration:

```
class String {
    ...
    static String valueOf( long l )  { ... }
    static String valueOf( float f ) { ... }
    ...
}
```

Anwendung/Aufruf:

```
String.valueOf( (float)(7./9.) )
```

liefert die Zeichenreihe: "0.7777778"

Beispiele: Klassenattribute, -methoden

Charakteristische Beispiele für Klassenattribute und -methoden liefert die Klasse `System`, die eine Schnittstelle von Programmen zur Umgebung bereitstellt:

```
class System {
  final static InputStream in  = ...;
  final static PrintStream out = ...;
  static void exit(int status)     { ... }
  static long currentTimeMillis() { ... };
}
```

Die Klasse `PrintStream` besitzt Methoden `print` und `println`:

```
System.out.print("Das erklaert die Syntax");
System.out.println(" von Printaufrufen");
```