TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

In der Vorlesung heute

- Dynamische Methodenauswahl
- Realisierung von Klassifikationen
- Exceptions

Wiederholung: Vererbung (engl. *inheritance*)

- Eine Klasse übernimmt Programmteile von einer anderen Klasse.
- Die erbende Klasse heißt **Subklasse**, die vererbende Klasse heißt **Superklasse**.
- In Java: Attribute und Methoden, **nicht** vererbt werden Konstruktoren sowie statische (Klassen)Attribute und statische (Klassen-)Methoden

Vererbung unterstützt Spezialisierung durch:

- Hinzufügen von Attributen (**Zustandserweiterung**)
- Hinzufügen von Methoden (**Erweiterung der Funktionalität**)
- Anpassen, Erweitern bzw. Reimplementieren von Supertyp-Methoden (**Anpassen der Funktionalität**)

Beispiel

```
interface Figure {
  double area();
  double distToOrigin();
}
abstract class AFigure implements Figure {
  protected CartPt loc;
  AFigure (CartPt loc)          { ... }
  public double distToOrigin() { ... }
}
class Square extends AFigure {
  private double size;
  Square (CartPt loc, double size) { ... }
  public double area()             { ... }
  public double getSize()          { ... }
}
class Circle extends AFigure {
  private double radius;
  Circle (CartPt loc, double radius) { ... }
  public double area()               { ... }
  public double getRadius()          { ... }
  public double distToOrigin()       { ... }
} ...
```

Vererbung und Information Hiding

Durch die Vererbung gibt es nun zwei Arten, eine Klasse K zu nutzen:

- *Anwendungsnutzung*: Erzeugen und Verwenden der Objekt von K
- *Vererbungsnutzung*: Spezialisieren und Erben von K

Damit die erbende Klasse die geerbten Programmteile geeignet nutzen kann, benötigt sie meist einen intimeren Zugriff als ein Anwendungsnutzer.

Modifikator `protected`: zugreifbar in allen Subklassen

Verschattung von vererbten Attributen

```
class C {
  public    int a = 0;
  public    int b = 2;
  private   int c = 3;
  int getC() {
    return c;
  }
}
```

```
class D extends C {
  public int e = 10;
  public int b = 12;
}
```

```
public class Zustandserweiterung {
  public static void main (String[] args) {
    D d = new D();
    System.out.println("Attribut e: " + d.e);
    System.out.println("Attribut b: " + d.b);

    System.out.println("Attribut a: " + d.a);
    System.out.println("Attribut b: " + ((C) d).b);
    System.out.println("Attribut c: " + d.getC());
  }
}
```

Attribute werden statisch, d.h. **während des Compilierens** gebunden.
Maßgebend ist also der (statische) Typ des selektierten Ausdrucks.

Dynamische Methodenauswahl bei Vererbung

Der **dynamische Typ** eines Objekts entspricht dem Klassentyp der Klasse, die bei der Erzeugung des Objektes angegeben wurde.

```
obj.m(...);
```

Beim Methodenaufruf wird die Implementierung der Methode basierend auf dem dynamischen Typ von `obj` ausgewählt.

Ist eine solche Implementierung in der Klasse des dynamischen Typs nicht vorhanden, wird die Implementierung jener Superklasse gewählt, die in der Vererbungshierarchie am weitesten "unten" liegt.

Beispiele: Dynamische Methodenauswahl I

1. Auswahl zwischen Methode der Super- und Subklasse:

```
AFigure c = new Circle (new CartPt(4.0,4.0), 1.0);
...
c.distToOrigin();
```

- Statische Typ der Variablen `c` ist `AFigure`.
- Dynamischer Typ des von `c` referenzierten Objekts ist `Circle`.

→ Implementierung der Methode `distToOrigin` aus der Klasse `Circle` wird ausgeführt

Beispiele: Dynamische Methodenauswahl II

2 Auswahl zwischen Methoden verschiedener Subklassen:

```
void getMaxArea (Figure[] df) {
  double maxArea = 0.0;
   for (int i = 0; i < df.length; i++) {
      Math.max(maxArea, df[i].area());
   }
   return maxArea;
}
```

- `df` hat Referenzen auf Objekte vom Typ `Figure`
- Verweist auf Objekte mit dynamischem Typ `Circle` oder `Square`
- Beim Aufruf der Methode `area` wird auf Basis dieses Typs entschieden, welche der beiden Implementierungen der Methode ausgeführt wird.

Beispiele: Dynamische Methodenauswahl III

3 Dynamische Methodenbindung auf `this`:

```
1   class A {
2     int f() {
3       return g() * 2;
4     }
5     int g() {
6       return 3;
7     }
8   }
9   class B extends A {
10    int g() {
11      return 21;
12    }
13  }
```

- Ausdruck `new B().f()` ruft Methode `f` in `A` auf
- Die Methode `f` ruft in Zeile 3 die Methode `g` auf (äquivalent zu `this.g()`)
- Dynamische Typ von `this` hier: `B` ⇒ Rufe `g` in `B` auf!
- Ergebnis von `new B().f()` ist 42

Frage: Was ist das Ergebnis?

```
class A {
   String a;
   A() {
     a = "aha";
     m();
   }
   void m(){
     System.out.print("Laenge a:" + a.length());
}  }

class B extends A {
   String b;
   B() {
      b = "boff";
      m();
   }
   void m() {
      System.out.print("Laenge b:" + b.length());
}  }

class KonstruktorProblemTest {
   public static void main( String[] args ){
      new B();
   }
}
```

Realisierung von Klassifikationen

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

Wann wählt man die Definition von Typen über Interfaces, wann über Klassen?

In der Regel

Verwendung von Schnittstellen in Java:

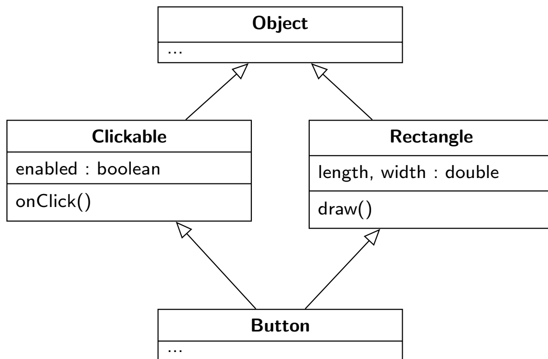
- keine Festlegung von Implementierungsteilen
- als Supertyp von Klassen mit mehreren Supertypen

Verwendung von Klassen in Java:

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

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.

Das Quadrat-Rechteck-Problem

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

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 {
  protected 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!
- Das Substitutionsprinzip erfordert, dass man Quadrat überall dort verwenden kann, wo ein Rechteck erwartet wird.
  Problem: Was ist die Semantik von `setWidth()` vs. `setLength()`

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

Ausnahmebehandlung mit Exceptions

Ausnahmebehandlung

Wie im Abschnitt zur “Terminierung” bereits angesprochen, kann die Auswertung eines Ausdrucks bzw. die Ausführung einer Anweisung:

- normal terminieren
- in eine Ausnahmesituation kommen und abrupt terminieren
- nicht terminieren

Klassifizierung von Ausnahmen

Es gibt drei Arten von Ausnahmesituationen:

1. Vom Programmierer schwer zu kontrollierende und zu beseitigende Situtationen (z.B. Speichermangel)
2. Programmierfehler (z.B. Null-Dereferenzierung, Verletzung von Indexgrenzen)
3. Zeitweise nicht verfügbare Ressourcen, anwendungsspezifische Ausnahmen, die behebbar sind.

Ausnahmen werden in Programmiersprachen verschieden behandelt:

- Programmabbruch (engl. *abort*)
- Ausnahmebehandlung

Ausnahmebehandlung in Java

Java bietet Sprachmittel für die **Ausnahmebehandlung** (engl. **exception handling**).

Dabei spielen drei Aspekte eine Rolle:

1. Wann/wie werden Ausnahmen ausgelöst?
2. Wie kann man sie abfangen?
3. Welcher Ausnahmetyp ist passend bzw. wie kann man neue Ausnahmetypen deklarieren?

Auslösen von Ausnahmen

Das Auslösen einer Ausnahme kann

- sprachdefiniert (z.B. NullPointer, IndexOutOfBounds) oder
- durch eine Anweisung spezifiziert sein.

In Java gibt es zum Auslösen von Ausnahmen die `throw`-Anweisung.

throw-Anweisung

Syntax:

Anweisung → `throw` Ausdruck ;

wobei der Ausdruck ein Ausnahmeobjekt als Ergebnis liefern muss.

Semantik:
Werte den Ausdruck aus.

Löst die Auswertung eine Ausnahme aus, ist dies die Ausnahme, die von der Anweisung ausgelöst wird.

Andernfalls löse die Ausnahme aus, die das Ergebnis des Ausdrucks ist.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Abfangen von Ausnahmen I

Die try-catch-Anweisung dient dem Abfangen und Behandeln von Ausnahmen:

```
3  void myMethod (String[] s) {
4     try {
5       System.out.println(s[0]);
6       System.out.println(s[1]);
7    } catch (NullPointerException e) {
8       System.out.println("s is null");
9    } catch (IndexOutOfBoundsException e){
10      System.out.println("s too small");
11   }
12 }
13
```

Abfangen von Ausnahmen II

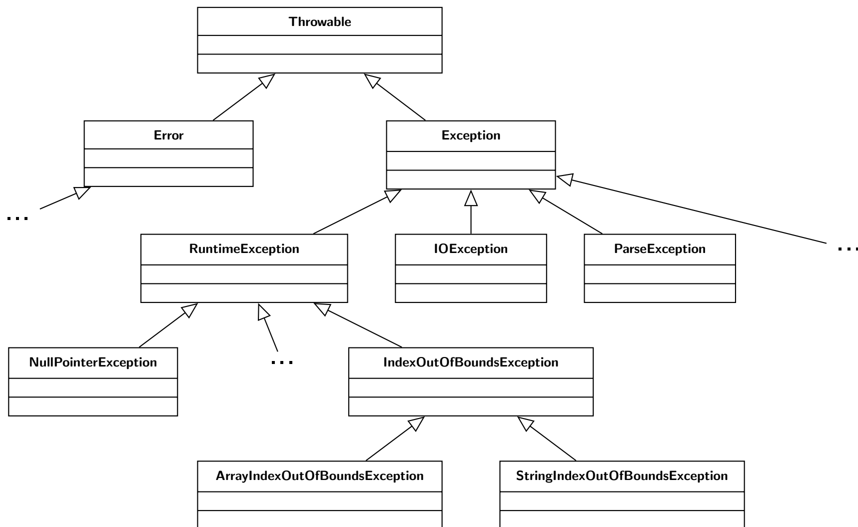
Tritt eine Ausnahme vom Typ T im try-Block auf, wird ein T-Objekt `x` erzeugt. Ist der Typ T in der Liste der catch-Klauseln aufgeführt,

- wird die Ausnahme *gefangen*,
- x an den Bezeichner der entsprechenden catch-Klausel gebunden und
- diese `catch`-Klausel ausgeführt

Objekt-orientierte Ausnahmebehandlung in Java

- Ausnahmen werden in Java durch Objekte repräsentiert; d.h. wenn eine Ausnahme auftritt, wird ein entsprechendes Ausnahmeobjekt erzeugt.
- Programmierer können eigene Exception-Klassen definieren.
- Sie müssen in die Typhierarchie der Java-Exceptions eingegliedert werden.

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 Exception {}
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
```

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.

```
3     public class UeberlaufException extends Exception {
4       UeberlaufException() {
5         super("Ueberlauf bei der Integer-Addition");
6       }
7     }
8     public class ExceptionTest {
9      public static void main(String[] args) {
10      try {
11       int ergebnis = 0;
12       int m = Integer.parseInt(args[0]);
13       int n = Integer.parseInt(args[1]);
14       long aux = (long) m + (long) n;
15       if (aux > Integer.MAX_VALUE) {
16          throw new UeberlaufException(); // selbstdefiniert
17       }
18       ergebnis = (int) aux;
19      } catch (ArrayIndexOutOfBoundsException e) {
20        System.out.println("Zuwenig Argumente");
21      } catch (Exception e) {
22        System.out.println("Ein Fehler ist aufgetreten");
23        e.getMessage();
24      }
25     }
26
```

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 }
23
```

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.

Ausgabe im Fehlerfall: Stacktraces

Die Ausführung führt zu einem Programmabbruch, weil die Ausnahme nicht in einem `try`-Block behandelt wurde.

Als Fehlermeldung wird ein sogenannter *Stacktrace* angezeigt:

```
Exception in thread "main" java.lang.Exception: Array darf nicht leer sein
    at ExceptionTest.checkPreconditions(ExceptionTest.java:20)
    at ExceptionTest.average(ExceptionTest.java:8)
    at ExceptionTest.main(ExceptionTest.java:5)
```

Vorteile von Exceptions

- Standardfall in der Ausführung von der Fehlerbehandlung syntaktisch getrennt
  $\Rightarrow$ **Exceptions nur zur Fehlerbehandlung verwenden!**
- Fehlerbehandlung wird an Aufrufer weitergegeben, da dieser evtl. besser reagieren kann
- Unterscheidung und Gruppierung von verschiedenen Fehlertypen möglich

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Klassenattribute und -methoden

Beispiel: Klassenattribute I

```
class Uebungsgruppe {
  static int maxTeilnehmer = 30;
  private String gruppenname;
  private Set<String> teilnehmer;

  Uebungsgruppe(String gruppenname){
    this.gruppenname = gruppenname;
    this.teilnehmer = new HashSet<String>();
  }  ...

  int freiePlaetze() {
    return maxTeilnehmer - teilnehmer.size();
  }
}
```

Beispiel: Klassenattribute II

```
public static void main (String[] args) {
    // Verwendung ohne Erzeugung von Objekten
    int i = Uebungsgruppe.maxTeilnehmer;

    Uebungsgruppe u = new Uebungsgruppe("Montagsgruppe");
    Uebungsgruppe u2 = new Uebungsgruppe("Dienstagsgruppe");
    System.out.println("Freie Plaetze montags: " + u.
    freiePlaetze());
    System.out.println("Freie Plaetze dienstags: " + u2.
    freiePlaetze());

    // Veraendern des Klassenattributs
    Uebungsgruppe.maxTeilnehmer = 32;
    System.out.println("Freie Plaetze montags: " + u.
    freiePlaetze());
    System.out.println("Freie Plaetze dienstags: " + u2.
    freiePlaetze());
}
```

Frage

Was machen wir, wenn Übungsgruppen unterschiedliche Größen haben sollen?

Klassenmethode

Klassenmethoden (statische Methoden) besitzen keinen impliziten Parameter (`this`).

Sie können nur auf Klassenattribute, ihre Parameter und ihre eigenen lokalen Variable zugreifen.

Beispiel: Klassenmethoden

Deklaration:

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

Anwendung/Aufruf:

```
    String.valueOf( (float)(7.0/9.0) )
```

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");
```

Geschachtelte Klassen

Begriffsklärung: Geschachtelte Klassen

Eine Klasse heißt in Java **geschachtelt** (engl. *nested*), wenn sie innerhalb der Deklaration einer anderen Klasse deklariert ist:

- als Komponente (ähnlich einem Attribut),
- als **lokale** Klassen (ähnlich einer lokalen Variablen)
- als **anonyme** Klassen bei der Objekterzeugung.

Ist eine Klasse nicht geschachtelt, nennen wir sie **global** (engl. *top-level*).

Beispiel: Statische Klassen

```
public class LinkedList {
    private Node entries = null;
    private int size = 0;

    private static class Node {
        int elem;
        Node next;
    }
    int getFirst() { ... }
    ...
}
```

Der Typ `Node` ist nur innerhalb von `LinkedList` sichtbar und zugreifbar. ⇒ Information Hiding

Statische geschachtelte Klassen

- Können erben und Interfaces implementieren und generisch sein.
  Achtung: Eine geschachtelte Klasse übernimmt **nicht** die Superklassen und Interfaces der umschließenden Klasse!
- Zugreifbar (falls sichtbar)über zusammengesetzte Namen von außerhalb (z.B. `Map.Entry` aus Java Collections).
- Zugriff auf private Attribute und Methoden der umfassenden Klasse (über Objektreferenz)