ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

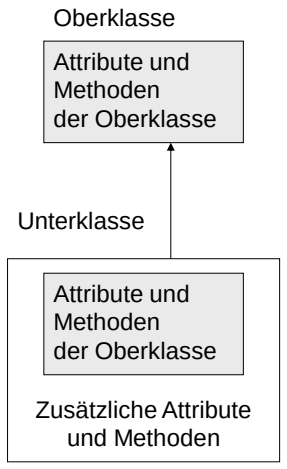
Vererbung

Prof. Dr. Helmut G. Folz

Vererbung

Definition:

- ***Vererbung*** ist ein Mechanismus, bei dem eine Klasse als Spezialfall einer allgemeinen Klasse definiert wird.
  - ⇨ Dabei "erbt" die ***Unterklasse*** automatisch alle Attribute und Methoden der ***Oberklasse***.
  - ⇨ Zusätzlich kann die Unterklasse weitere Attribute und Methoden hinzufügen und geerbte Methoden redefinieren.

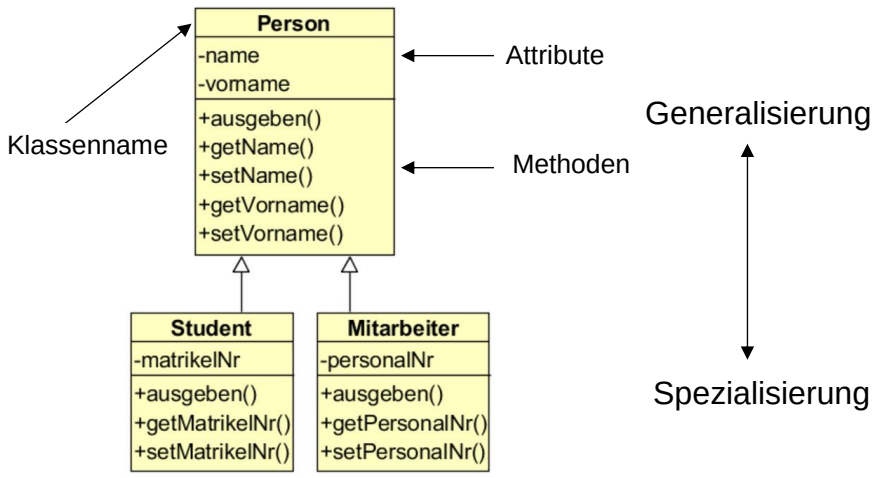


Beispiel

- Für ein Hochschulinformationssystem wird eine Klasse für Mitarbeiter und eine Klasse für Studenten benötigt. Die Analyse ergibt die folgenden benötigten Merkmale:

Klasse `Mitarbeiter`:	Klasse `Student`:
Attribute: • Name • Vorname • Personalnummer Methoden: • set- und get-Methoden • Ausgabe auf die Standardausgabe	Attribute: • Name • Vorname • Matrikelnummer Methoden: • set- und get-Methoden • Ausgabe auf die Standardausgabe

Beispiel



Klasse Person (1)

```
package person1;
public class Person {
    /**
     * Person auf die Standardausgabe ausgeben
     *
     */
    public void ausgeben() {
        System.out.print(name + ", " + vorname);
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setVorname(String vorname) {
        this.vorname = vorname;
    }
```


Klasse Person (2)

```
    public String getName() {
        return name;
    }

    public String getVorname() {
        return vorname;
    }

    @Override
    public String toString() {
        return name + ", " + vorname;
    }

    private String name;
    private String vorname;
}
```

Klasse Mitarbeiter (1)

```
package person1;
public class Mitarbeiter extends Person {
    /**
     * Mitarbeiter auf die Standardausgabe ausgeben
     *
     */
    @Override
    public void ausgeben() {
        super.ausgeben();
        System.out.print("\tPers-Nr: " + personalNr);
    }

    public void setPersonalNr(int personalNr) {
        this.personalNr = personalNr;
    }

    public int getPersonalNr() {
        return personalNr;
    }
```

Ordne die Klasse einem Paket zu

Aufruf der entsprechenden Methode der Oberklasse

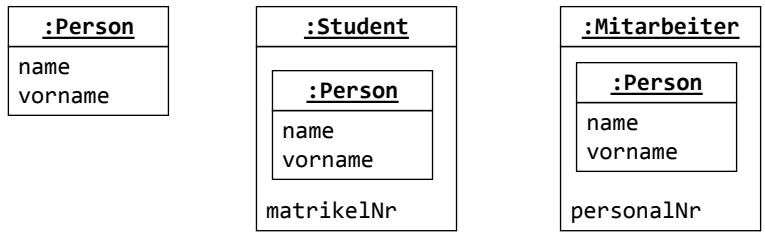
Klasse Mitarbeiter (2)

```
    @Override
    public String toString() {
        return super.toString() + "\tPers-Nr: " + personalNr;
    }

    private int personalNr;
}
```

Klasse Student

```
package person1;
public class Student extends Person {
    @Override
    public void ausgeben() {
        super.ausgeben();
        System.out.print("\tMat-Nr: " + matrikelNr);
    }
    public void setMatrikelNr(int matrikelNr) {
        this.matrikelNr = matrikelNr;
    }
    public int getMatrikelNr() {
        return matrikelNr;
    }
    @Override
    public String toString() {
        return super.toString() + "\tMat-Nr: " + matrikelNr;
    }

    private int matrikelNr;
}
```


Innerer Aufbau (in etwa)



Wichtige Eigenschaften (1)

- Die **Unterklasse** (abgeleitete Klasse) erbt von der **Oberklasse** (Basisklasse) alle Attribute und Methoden
- Zusätzliche Attribute und Methoden können hinzugefügt werden
- Geerbte Methoden können redefiniert werden.
  - ⇨ Dabei muss die Methode der Unterklasse die gleiche Signatur (Name + Anzahl + Typ der Parameter) wie die geerbte Methode haben.
- Geerbte Attribute können nicht redefiniert werden
- Die Unterklasse hat Zugriff auf alle `public`- und `protected`-Merkmale sowie auf alle *package*-Merkmale, sofern die Oberklasse zum gleichen Paket gehört

Wichtige Eigenschaften (2)

- Die privaten Merkmale der Basisklasse werden zwar vererbt, jedoch ist ein direkter Zugriff in der abgeleiteten Klasse nicht erlaubt.
- Klassen können an beliebig viele Klassen weitervererben.
- Eine Klasse kann bei Java nur von genau einer Oberklasse erben (*Einfachvererbung*).
- Unterklassen können wieder als Basisklassen dienen, so dass regelrechte Klassenhierarchien entstehen können.
- Jede Klasse hat genau eine Oberklasse.
  - ⇨ Ist keine explizite Oberklasse angegeben, so ist dies automatisch die Klasse `java.lang.Object`.

Testprogramm (1)

```
public class PersonTest1 {

    public void start() {
        Person p1 = new Person();
        p1.setName("Müller");
        p1.setVorname("Thomas");
        p1.ausgeben();
        System.out.println();

        Student s1 = new Student();
        s1.setName("Reus");
        s1.setVorname("Marco");
        s1.setMatrikelNr(1234567);
        s1.ausgeben();
        System.out.println();
```


Testprogramm (1)

```
        Mitarbeiter m1 = new Mitarbeiter();
        m1.setName("Löw");
        m1.setVorname("Jogi");
        m1.setPersonalNr(4711);
        System.out.println(m1);
    }

    public static void main(String[] args) {
        new PersonTest1().start();
    }
}
```

Ist-ein(e)- und hat-ein(e)-Beziehung

- Zwischen der Unterklasse und der Oberklasse besteht eine sogenannte *ist-ein(e)-Beziehung*,
  - ⇨ d. h. ein Objekt der Unterklasse ist ein spezielles Objekt der Oberklasse.
- Eine andere Art der Beziehung zwischen zwei Klassen ist die *hat-ein(e)-Beziehung*.
  - ⇨ Zwei Klassen stehen in einer *hat-ein(e)-Beziehung*, wenn eine Klasse ein Objekt der anderen als Element besitzt bzw. eine Referenz auf ein Objekt der anderen Klasse besitzt.

Attribute und Methoden

- Attribute werden von der Oberklasse an die Unterklasse weitervererbt und können nicht redefiniert werden.
- Attribute können allerdings durch gleichnamige Attribute in der Unterklasse überdeckt werden.
- Methoden können in Unterklassen redefiniert werden. Dazu müssen Signatur und Rückgabetyp gleich sein.

Attribute und Methoden

```
public class RedefOben {
    protected String str = "x";

    public void f() {
        System.out.println("RedefOben.f()");
    }
}

public class RedefUnten extends RedefOben {
    private String str = "y"; // Überdeckung

    @Override
    public void f() {  // Redefinition
        System.out.println("RedefTestUnten.f()");
    }

    public void f(int i) {    // Neue Methode
        System.out.println("RedefTestUnten.f(int)");
    }
}
```


Annotationen

- Seit Java 5 gibt es in der Programmiersprache Java **Annotationen** (engl. annotations).
- Annotationen bieten die Möglichkeit, sogenannte Meta-Daten im Code unterzubringen.
- Annotationen beeinflussen nicht direkt die Programmsemantik, sie beeinflussen jedoch die Art und Weise wie Programme von Tools und Bibliotheken behandelt werden.
- Annotationen können aus den Quell-Dateien, aus class-Dateien und zur Laufzeit gelesen werden.

Die Annotation @Override

@Override

- Mit diesem Annotationstyp aus dem Package java.lang kann eine Methode gekennzeichnet werden, die die Methode ihrer Oberklasse überschreibt.
- Der Compiler stellt dann sicher, dass die Oberklasse diese Methode enthält und gibt einen Fehler aus, wenn dies nicht der Fall ist.

@Override: Beispiel

```
class Food {}
class Hay extends Food {}
class Animal {
    Food getPreferredFood() { return null; }
}

class Horse extends Animal {
    Horse() { return; }
    @Override
    Hay getPreferredFood() { return new Hay(); }
}
```

- Der Return-Typ der redefinierten Methode ist von einem Unterklassentyp des Return-Typs der Original-Methode!
- Das ist nur mit Hilfe der Annotation @Override realisierbar.

Das Schlüsselwort super

- **super** ist in allen nicht klassenbezogenen Methoden einer Unterklasse verfügbar.
- **super** stellt eine Referenz zum aktuellen Objekt als ein Exemplar seiner Oberklasse dar.
  - ⇨ `super.str` Zugriff auf Attribut `str` der Klasse `RedefOben`
  - ⇨ `super.f()` Zugriff auf Methode `f()` der Klasse `RedefOben`

Das Schlüsselwort final

- Mit dem Schlüsselwort **final** gekennzeichnete Methoden können nicht redefiniert werden:

```
public final String getName() {
   return name;
}
```

- Mit Hilfe dieses Schlüsselwortes kann sogar das Erben von einer bestimmten Klasse verboten werden.

```
public final class String {
   ...
}
```

Konstruktoren (1)

```
package person2;

/**
 * Personenklasse erweitert um Konstruktoren
 * @author folz
 */
public class Person {
    public Person() {}

    public Person(String name, String vorname) {
        this.name = name;
        this.vorname = vorname;
    }
    ...
}
```


Konstruktoren (2)

```
public class Mitarbeiter extends Person {
    /**
     * Standardkonstruktor, ruft implizit den
     * Standardkonstruktor der Oberklasse auf
     */
    public Mitarbeiter() {}
    /**
     * Konstruktor, der direkt den Konstruktor der
     * Oberklasse aufruft.
     */
    public Mitarbeiter (String name, String vorname,
                        int personalNr) {
        super(name, vorname);
        this.personalNr = personalNr;
    }
    ...
}
```

Konstruktoren (3)

```
public class Student extends Person {
    /**
     * Standardkonstruktor, der den Standardkonstruktor
     * der Oberklasse direkt aufruft.
     */
    public Student() {}

    public Student (String name, String vorname,
                    int matrikelNr) {
        super(name, vorname);
        this.matrikelNr = matrikelNr;
    }
    ...
}
```


Konstruktoren (4)

```
package person2;
public class PersonTest2 {
    public void start() {
        Person p1 = new Person("Müller", "Thomas");
        p1.ausgeben();
        System.out.println();

        Student s1 = new Student("Reus", "Marco", 1234567);
        s1.ausgeben();
        System.out.println();

        Mitarbeiter m1 = new Mitarbeiter("Löw", "Jogi", 4711);
        m1.ausgeben();
        System.out.println();
    }

    public static void main(String[] args) {
        new PersonTest2().start();
    }
}
```

Ablauf der Objekterzeugung

Bei der Erzeugung eines Objektes wird immer die folgende Reihenfolge eingehalten:

- Aufruf des Konstruktors der Unterklasse; dort direkt Aufruf des Konstruktors der Oberklasse
  - ⇨ Attribute auf voreingestellte Anfangswerte setzen (`0` für numerische Typen, `\u0000` für `char`, `false` für `boolean`, `null` für Referenztypen)
  - ⇨ Initialisierung der Attribute mittels ihrer Initialisierungsausdrücke
  - ⇨ Aufruf expliziter Initialisierungsblöcke
  - ⇨ Ausführung des Konstruktorrumpfes der Oberklasse
- Der gleiche Ablauf in der Unterklasse:
  - ⇨ Initialisieren und Ausführen Konstruktorrumpf

Ablauf der Objekterzeugung (1)

```
public class Oberklasse {
    protected int i = 1;
    {   // Objektinitialisierungsblock
        System.out.println("Oberklasse: Initialisierungsblock");
        System.out.println("vorher  i = " + i );
        i = 2;
        System.out.println("nachher i = " + i );
    }

    public Oberklasse() {
        System.out.println("Oberklasse: Konstruktor ");
        System.out.println("vorher  i = " + i );
        i = 3;
        System.out.println("nachher i = " + i );
    }
}
```

Ablauf der Objekterzeugung (2)

```
public class Unterklasse extends Oberklasse {
    private int j = 1;
    {   // Objektinitialisierungsblock
        System.out.println("Unterklasse: Initialisierungsblock");
        System.out.println("vorher  i = " + i + " j = " + j);
        j = 2;
        System.out.println("nachher i = " + i + " j = " + j);
    }
    public Unterklasse() {
        System.out.println("Unterklasse: Konstruktor ");
        System.out.println("vorher  i = " + i + " j = " + j);
        j = 3;
        System.out.println("nachher i = " + i + " j = " + j);
    }

    static public void main(String[] args) {
        Unterklasse u = new Unterklasse();
    }
}
```


Ablauf der Objekterzeugung (3)

```
/* Ausgabe:
Oberklasse: Initialisierungsblock
vorher  i = 1
nachher i = 2
Oberklasse: Konstruktor
vorher  i = 2
nachher i = 3
Unterklasse: Initialisierungsblock
vorher  i = 3 j = 1
nachher i = 3 j = 2
Unterklasse: Konstruktor
vorher  i = 3 j = 2
nachher i = 3 j = 3

*/
```

Typkonvertierungen

- Wie bei den elementaren Datentypen gibt es auch bei Referenztypen feste Regeln für Typkonvertierungen.
- Wir betrachten hierzu unsere Personenhierarchie:

```
Person      p1, p2;         // Person-Referenzen
Student     s1 = new Student();
                 // Student-Objekt
Mitarbeiter m1 = new Mitarbeiter();
                 // Mitarbeiter-Objekt

p1 = s1;
p2 = m1;
```

- Was passiert hier?

Explizite Typkonvertierung

- Betrachte:

```
Person p1 = new Student();     // erlaubt!
Student s2 = (Student)p1;      // erlaubt! Warum?
```

- Eine solche explizite Typkonvertierung ist nur erlaubt in einer Klassenhierarchie
  - ⇨ als sogenannter *Down-Cast*, also eine Konvertierung in einen Typ, der in der Hierarchie weiter unten liegt
  - ⇨ oder als *Up-Cast*, also aufwärts in der Hierarchie.
- Generell gilt: Eine explizite Typkonvertierung konvertiert nicht den Typ des Objekts, sondern nur die Referenz darauf.

Der Operator instanceof

- Syntax:

  a `instanceof` *Klassenname*

- Dieser Ausdruck hat den Wert `true`, wenn die Referenz *a* auf ein Objekt der Klasse *Klassenname* bzw. auf ein Objekt, dessen Klasse von der Klasse *Klassenname* abgeleitet ist, zeigt.
- Wenn a die Nullreferenz ist, hat der Ausdruck immer den Wert `false`

Der Operator instanceof

```
p1 = new Mitarbeiter();

if (p1 instanceof Person)// true
    System.out.println("p1 ist ein Person-Objekt");

if (p1 instanceof Mitarbeiter)  // true
    System.out.println("p1 ist ein Mitarbeiter-Objekt");

if (p1 instanceof Student) {    // false
    System.out.println("p1 ist ein Student-Objekt");
    s1 = (Student) p1;
}
```

Polymorphismus

- Ein gegebenes Programmelement kann sich zur Laufzeit auf Objekte ganz verschiedener Klassen beziehen.
- Methoden von Objekten können unter gleichem Namen angesprochen werden, aber erst zum Zeitpunkt des Programmablaufs muss feststehen,
  - ⇨ zu welcher Klasse das Objekt gehört,
  - ⇨ welche Operation tatsächlich zur Ausführung kommt.
- Weil auszuführende Operationen erst zur Laufzeit "gebunden" werden, heißt diese Technik auch ***dynamisches Binden***.

Beispiel: Polymorphismus (1)

```
public class PersonTest4 {
    public void start() {
        Person [] pTab = new Person[5];
        pTab[0] = new Person("Schmitt", "Hans");
        pTab[1] = new Student("Meier", "Fritz", 1111111);
        pTab[2] = new Student("Hoffmann", "Petra", 2222222);
        pTab[3] = new Mitarbeiter("Adam", "Albert", 4711);
        pTab[4] = new Mitarbeiter("Beyer", "Gerda", 4712);

        for (int i = 0; i < pTab.length; ++i) {
            pTab[i].ausgeben();
            System.out.println();
        }
    }

    public static void main(String[] args) {
        new PersonTest4().start();
    }
}
```

Beispiel: Polymorphismus (2)

```
/* Ausgabe:
Schmitt, Hans
Meier, Fritz    Mat-Nr: 1111111
Hoffmann, Petra Mat-Nr: 2222222
Adam, Albert    Pers-Nr: 4711
Beyer, Gerda    Pers-Nr: 4712
*/
```


Statisches Binden

Bei Java werden nur die folgenden Methoden statisch gebunden:

- Klassenmethoden (Zusatz: `static`)
  - ⇨ Klassenmethoden können nicht redefiniert werden, sondern höchstens durch eine Klassenmethode mit gleicher Signatur überdeckt werden.
- Finale Methoden (Zusatz: `final`)
  - ⇨ Finale Methoden können nicht refediniert werden, daher macht dynamisches Binden für Sie keinen Sinn.
- Private Methoden (Zusatz: `private`)
  - ⇨ Private Methoden werden zwar an Unterklassen weitervererbt, können dort aber nicht aufgerufen werden.
  - ⇨ Da sie sowieso nur in der Klasse, in der sie definiert sind aufgerufen werden können, werden sie ebenfalls statisch gebunden.

Konstruktor und dynamisches Binden (1)

- Betrachten Sie das folgende Beispiel:

```
public class Oben {
    protected int a;

    public Oben() {
        System.out.println("Oben Konstruktor");
        init(); // Aufruf einer public-Methode!
    }

    public void init() {
        System.out.println("Oben.init()");
        a = 1;
    }

    public String toString() {
        return "Oben: a = " + a;
    }
}
```


Konstruktor und dynamisches Binden (1)

```
public class Unten extends Oben {
    protected int b;
    public Unten() {
        System.out.println("Unten Konstruktor");
        init();
    }
    public void init() {
        System.out.println("Unten.init()");
        b = 2;
    }

    public String toString() {
        return super.toString() + "\nUnten: b = " + b;
    }

    public static void main (String[] args) {
        System.out.println("Unten.main() Start");
        Unten u = new Unten();
        System.out.println(u);
        System.out.println("Unten.main() Ende");
    }
}
```

Was passiert hier ?

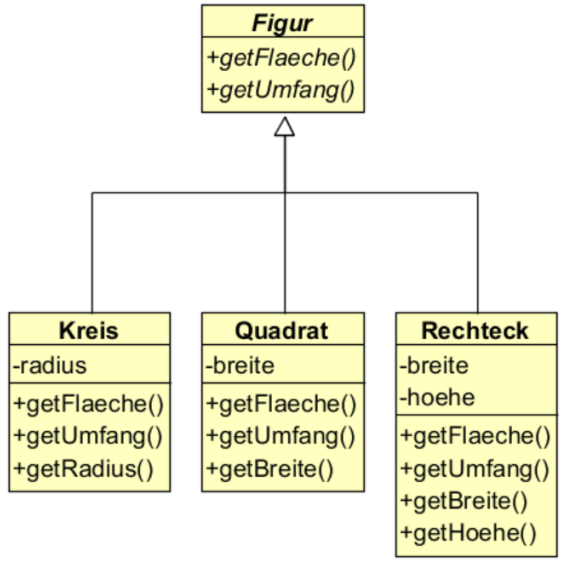
Konstruktor und dynamisches Binden (2)

- Ausgabe:

```
Unten.main() Start
Oben Konstruktor
Unten.init()
Unten Konstruktor
Unten.init()
Unten.main() Ende
```

- Offensichtlich wird im Konstruktor der Oberklasse nicht die `init`-Methode der Oberklasse aufgerufen sondern die `init`-Methode der Unterklasse, d. h. schon im Konstruktor wird dynamisch gebunden.
- Übrigens ist das bei C++ so nicht möglich.
- Lösung: definiere `init()` in der Oberklasse `private`. Dann wird nicht dynamisch gebunden

Beispiel: Abstrakte Klassenhierarchie (1)



Beispiel: Abstrakte Klassenhierarchie (2)

- Problem: `getFlaeche()` ist in der Klasse `Figur` nicht implementierbar !
- Lösung: `getFlaeche` in Figur als abstrakte Methode deklarieren, d. h. als Methode ohne Implementierung.
- Eine abstrakte Methode wird in Java mit Hilfe des Schlüsselwortes `abstract` definiert.
- Eine abstrakte Methode besitzt keine Implementierung.
- Eine Klasse, die eine abstrakte Methode enthält, muss ebenfalls mit dem Zusatz `abstract` versehen werden und ist damit eine abstrakte Klasse.

Beispiel: Abstrakte Klassenhierarchie (3)

```
public abstract class Figur {
    /**
     * Flaeche berechnen
     *
     * @return Flaeche
     */
    public abstract double getFlaeche();

    /**
     * Umfang berechnen
     *
     * @return Umfang
     */
    public abstract double getUmfang();
}
```

Beispiel: Abstrakte Klassenhierarchie (4)

```
public class Kreis extends Figur {
    protected double r = 1.0;
    public Kreis() {}

    public Kreis(double r) {
        this.r = r;
    }

    public double getFlaeche() {
        return Math.PI * r * r;
    }

    public double getUmfang() {
        return 2 * Math.PI * r;
    }

    public double getRadius() {
        return r;
    }
}
```


Beispiel: Abstrakte Klassenhierarchie (5)

```
public class Quadrat extends Figur {
    protected double breite = 0.0;

    public Quadrat(double b) {
        breite = b;
    }

    public double getFlaeche() {
        return breite * breite;
    }

    public double getUmfang() {
        return 4 * breite;
    }

    public double getBreite() {
        return breite;
    }
}
```

Eigenschaften von abstrakten Klassen

- Von einer abstrakten Klasse können keine Objekte erzeugt werden. Sie kann nur als Oberklasse für konkrete Klassen dienen.
- Referenzen auf abstrakte Klassen sind zulässig und werden auch sehr häufig eingesetzt.
  - ⇨ Eine solche Referenz darf natürlich auf Objekte von konkreten Unterklassen verweisen.
- Eine von einer abstrakten Oberklasse geerbte abstrakte Methode muss in der Unterklasse implementiert werden andernfalls ist die Unterklasse selbst wiederum abstrakt.

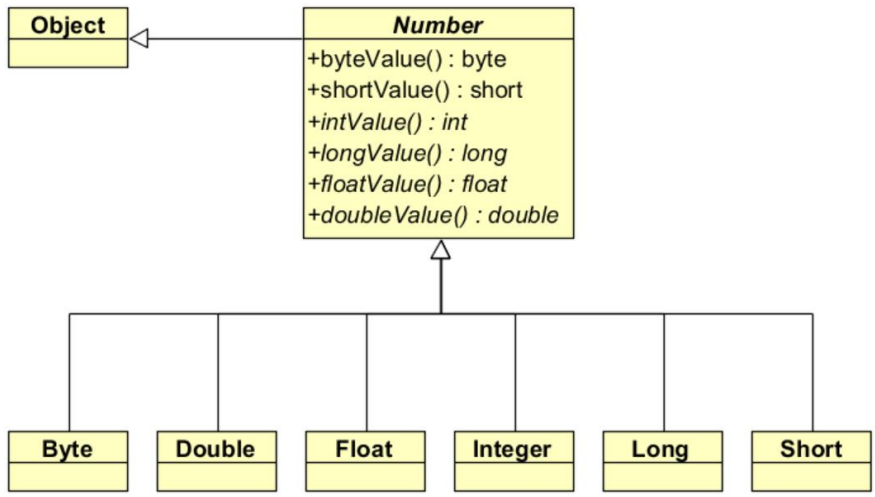
Testprogramm Figur-Hierarchie (1)

```
public class FigurTest {
    public void start() {
        Figur [] figurTab = new Figur[3];
        figurTab[0] = new Kreis(3.0);
        figurTab[1] = new Rechteck(1.0,2.0);
        figurTab[2] = new Quadrat(2.0);

        double gesamtFlaeche = 0.0;
        for (int i = 0; i < figurTab.length; i++) {
            System.out.println("figurTab[" + i + "].getFlaeche(): "
                + figurTab[i].getFlaeche());
            System.out.println("figurTab[" + i + "].getUmfang() : "
                + figurTab[i].getUmfang());
            gesamtFlaeche += figurTab[i].getFlaeche();
        }
        System.out.println("gesamtFlaeche: " + gesamtFlaeche);
    }

    public static void main(String[] args) {
        new FigurTest().start();
    }
}
```

Testprogramm Figur-Hierarchie (2)

```
/* Ausgabe:
figurTab[0].getFlaeche(): 28.274333882308138
figurTab[0].getUmfang() : 18.84955592153876
figurTab[1].getFlaeche(): 2.0
figurTab[1].getUmfang() : 6.0
figurTab[2].getFlaeche(): 4.0
figurTab[2].getUmfang() : 8.0
gesamtFlaeche: 34.27433388230814
*/
```


Klassenhierarchie aus java.lang

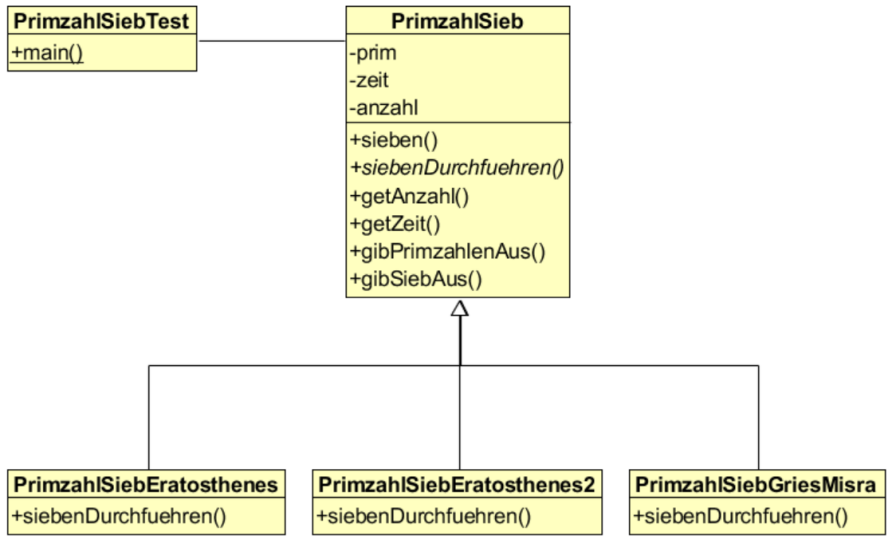


Rahmen für Primzahlsiebe (1)

Verschiedene Primzahlsiebe sollen ausgetestet werden können.

- Sieb des Eratosthenes (naiver Ansatz)
- Sieb des Eratosthenes (verbesserter Ansatz)
- Sieb von Gries und Misra (1978)

Rahmen für Primzahlsiebe (2)



Rahmen für Primzahlsiebe (3)

```
public class PrimzahlSiebTest {
    public static void main(String [] args) {
        Scanner input = new Scanner(System.in);
        PrimzahlSieb ps;
        int verfahren;
        System.out.print("Obere Grenze: ");
        int max = input.nextInt();
        do {
            ps = null;
            System.out.print("Verfahren: 1 = Eratosthenes / "
                                    + "2 = Eratosthenes2 / "
                                    + "3 = Gries/Misra / "
                                    + "0 = Beenden: ");
            verfahren = input.nextInt();
            switch(verfahren) {
            case 1: ps = new PrimzahlSiebEratosthenes(max);
                    break;
            case 2: ps = new PrimzahlSiebEratosthenes2(max);
                    break;
            case 3: ps = new PrimzahlSiebGriesMisra(max);
                    break;
            }
```


Rahmen für Primzahlsiebe (4)

```
            if (ps != null) {
                ps.sieben();
                ps.gibSiebAus();
                System.out.print("Primzahlen ausgeben (j/n) ? ");
                if (input.next().charAt(0) == 'j')
                    ps.gibPrimzahlenAus();
            }
        } while (verfahren != 0);
    }
}
```

Rahmen für Primzahlsiebe (5)

```
/**
 * Abstraktes Primzahlsieb
 */
public abstract class PrimzahlSieb {
    protected boolean[] prim;
    protected long zeit;  // gemessene Zeit
    protected int anzahl; // Anzahl Primzahlen

    /**
     * Konstruiere das boolean-Feld, mit dessen Hilfe das Sieb
     * durchgefuehrt werden soll
     *
     * @param max Groesse des Feldes
     */
    public PrimzahlSieb(int max) {
        prim = new boolean[max+1];
        for (int i = 0; i < prim.length; i++)
            prim[i] = true;
        prim[0] = prim[1] = false;
    }
```


Rahmen für Primzahlsiebe (6)

```
    /**
     * Durchfuehren des Primzahlsieb-Algorithmus mit Messen der
     * Zeit
     */
    public void sieben()  {
        zeit = System.currentTimeMillis();
        siebenDurchfuehren();
        zeit = System.currentTimeMillis() - zeit;
    }

    /**
     * Vorgabe fuer den eigentlichen Sieb-Algorithmus
     */
    public abstract void siebenDurchfuehren();

    /**
     * Ermittelte Zeit zurueckgeben
     *
     * @return ermittelte Zeit
     */
    public long getZeit() {
        return zeit;
    }
```

Rahmen für Primzahlsiebe (7)

```
/**
     * Ermittelte Anzahl an Primzahlen zurueckgeben
     * @return Anzahl Primzahlen im Intervall
     */
    public int getAnzahl() {
        if (anzahl == 0) {
            for (int i = 2; i < prim.length; i++)
                if (prim[i])
                    anzahl++;
        }
        return anzahl;
    }


    /**
     * Ausgeben der ermittelten Ergebnisse
     */
    public void gibSiebAus() {
        System.out.println("Primzahlen bis " + (prim.length-1));
        System.out.println("Benötigte Rechenzeit: " + zeit + " ms");
        System.out.println(getAnzahl() + " Primzahlen insgesamt");
    }
```


Rahmen für Primzahlsiebe (8)

```
public void gibPrimzahlenAus() {
    int zaehl = 0;
        for (int i = 2; i < prim.length; i++) {
            if (prim[i]) {
                System.out.print(i + "\t");
                zaehl++;
                if (zaehl%10 == 0)
                    System.out.println();
            }
        }
        System.out.println();
    }
}
```

Naiver Eratosthenes

```
/**  Primzahlsieb nach Eratosthenes (einfache Version) /
public  class PrimzahlSiebEratosthenes extends PrimzahlSieb {
    public PrimzahlSiebEratosthenes(int max) {
        super(max);
    }

    /**
     * Durchfuehrung des Sieb-Algorithmus nach einem sehr simpel
     * gestrickten Verfahren
     */
    public void siebenDurchfuehren () {
        int i, j;
        int max = prim.length - 1;  // maximal zu untersuchende Zahl

        for (i = 2; i <= max; i++ ){
            for (j = 2; j <= max /i; j++) {
                prim[i*j] = false;
            }
        }
    }
}
```


Verbesserter Eratosthenes

```
/** Primzahlsieb nach Eratosthenes (verbesserte Version)   */
public  class PrimzahlSiebEratosthenes2 extends PrimzahlSieb {
    public PrimzahlSiebEratosthenes2(int max) {
        super(max);
    }
    /**
     * Durchfuehrung des Sieb-Algorithmus unter Vermeidung
     * unnoetiger Redundanzen
     */
    public void siebenDurchfuehren () {
        int i, j;
        int max = prim.length - 1;
        int grenze = (int)Math.round(Math.sqrt(max));

        for (i = 2; i <= grenze; i++ ){
            if (prim[i]) {
                for (j = 2; j <= max/i; j++)
                    prim[i*j] = false;
            }
        }
    }
}
```

Primzahlsieb nach Gries und Misra (1)

```
/** Primzahlsieb nach Gries und Misra (1978)  */
public  class PrimzahlSiebGriesMisra extends PrimzahlSieb {
    public PrimzahlSiebGriesMisra(int max) {
        super(max);
    }
    /** Durchfuehrung des Sieb-Algorithmus nach dem Algorithmus
     *  von Gries und Misra (1978)
     */
    public void siebenDurchfuehren () {
        long p, q, x;
        int max = prim.length - 1; // maximal zu untersuchende Zahl

        // Fuer alle Primzahlen bis sqrt(max)
        for (p = 2; p*p <= max; p = next(p)) {
            // bilde alle Produkte p, p*p, p*p*p
            // und q*p, q*p*p, ... fuer enthaltene Primzahlen q
            for (q = p; p*q <= max; q = next(q)) {
            // Multipliziere Ausdruck mit p und streiche ihn heraus
                for (x = p*q; x <= max; x *= p)
                    prim[(int)x] = false;
            }
        }
    }
```

Primzahlsieb nach Gries und Misra (2)

```
    /**
     * Naechste nicht gestrichene Zahl in der Siebtabelle finden
     *
     * @param p Primzahl, von der aus gesucht wird.
     */
    public long next(long p) {
        long nextPrime = p+1;
        while (!prim[(int)nextPrime])
            nextPrime++;
        return nextPrime;
    }
}
```

Interfaces

- Einfachvererbung ist oft nicht ausreichend bei der Modellierung objektorientierter Systeme.
- Mehrfachvererbung ist allerdings häufig problematisch für Compiler und Entwickler gleichermaßen.
- Ein Interface (Schnittstellenklasse) beschreibt speziell ausgewählte Eigenschaften für Klassen
  - ⇨ alle Methoden sind abstrakt
  - ⇨ Methoden können nicht `static` sein (bis Java 8!)
  - ⇨ alle Attribute sind automatisch `final` und `static`
  - ⇨ alle Merkmale sind implizit `public`
- Ein Interface entspricht also prinzipiell einer abstrakten Klasse ohne Objektattribute und mit lauter abstrakten Methoden.
- Eine Klasse kann beliebig viele Schnittstellen implementieren, d.h. die vorgegebenen abstrakten Methoden werden realisiert.

Interfaces ab Java 8

- Für abstrakte Methoden können Default-Implementierungen angegeben werden.
  - ⇨ Da es aber keine Objektattribute geben kann, gibt es nicht sehr viele sinnvolle Anwendungsmöglichkeiten
- Interfaces können jetzt auch Klassenmethoden enthalten, die natürlich eine Implementierung haben müssen, da es keine abstrakten Klassenmethoden gibt.
  - ⇨ Da es nur final-Klassenattribute geben kann, kann hier höchstens lesend zugegriffen werden.

java.lang.Comparable

```
// Veraltete Darstellung
public interface Comparable {
    public abstract int compareTo(Object o);
}
```

Das Interface `Comparable` wird z. B. implementiert durch die Klassen `java.lang.String`, sowie die numerischen Hüllenklassen `java.lang.Byte`, ...

```
public final class Byte extends Number
                         implements Comparable { ... }
```

oder

```
public final class String implements Serializable,
                                     Comparable { ... }
```


Anwendung auf Person

```
public class Person implements Comparable {
  ...

   /**
    * Vergleicht das Objekt mit dem uebergebenen;
    *
    * @return liefert 0 bei Gleichheit, 1 falls das
    * Objekt groesser, -1 falls das Objekt kleiner ist
    */
   public int compareTo(Object o) {
       Person tmp = (Person)o;
       int ret = name.compareTo(tmp.name);
       if (ret == 0)
           return vorname.compareTo(tmp.vorname);
       else
           return ret;
   }

 ...
```

Anwendung von Comparable

```
Comparable c1 = "abcd",       // c1 und c2 dürfen auf
           c2 = "abcde";      // String-Objekte zeigen

if ( c1.compareTo(c2) < 0)    // einzig aufrufbare Methode
    System.out.println("c1 kleiner als c2");
```

```
Comparable c1 = new Person("Meier", "Albert"),
           c2 = new Person("Meier", "Berthold");

if ( c1.compareTo(c2) < 0)    // einzig aufrufbare Methode
    System.out.println("c1 kleiner als c2");
```


ComparableTest (1)

```
public class ComparableTest {
    /**
     *  Universeller Sortieralgorithmus, der alle Felder
     *  von Comparable-Typen sortiert
     *
     *  @param t zu sortierendes Array
     */
    public static void bubbleSort(Comparable[] t) {
        int i, j;
        Comparable tmp;
        for (i = 0; i < t.length; i++)  {
           for (j = i + 1; j < t.length; j++) {
              if (t[i].compareTo(t[j]) > 0) {
                  tmp = t[i];
                  t[i] = t[j];
                  t[j] = tmp;
              }
           }
        }
    }
```

ComparableTest (2)

```
public static void teste() {
        String[] stab = {"Emil", "August", "Peter",
                         "Agathe", "Lisa"};
        print(stab);
        bubbleSort(stab);
        print(stab);

        Person [] pTab = new Person[5];
        pTab[0] = new Person("Schmitt" , "Hans");
        pTab[1] = new Person("Meier"   , "Fritz");
        pTab[2] = new Person("Hoffmann", "Petra");
        pTab[3] = new Person("Adam"    , "Albert");
        pTab[4] = new Person("Beyer"   , "Gerda");
        print(pTab);
        bubbleSort(pTab);
        print(pTab);
    }
```


ComparableTest (3)

```
    public static void print(Comparable[] t) {
        for (int i = 0; i < t.length; i++)
            System.out.print(t[i] + " | ");
        System.out.println();
    }

    public static void main(String args[]) {
        teste();
    }
}
```

ComparableTest (4)

```
Emil | August | Peter | Agathe | Lisa |
Agathe | August | Emil | Lisa | Peter |

Schmitt, Hans | Meier, Fritz | Hoffmann, Petra | Adam,
Albert | Beyer, Gerda |
Adam, Albert | Beyer, Gerda | Hoffmann, Petra | Meier, Fritz
| Schmitt, Hans |
```


Erweiterung von Interfaces

```
interface V { ... }
interface W extends V { ... }    // einfach

interface X { ... }
interface Y extends W, X { ... } // und mehrfach möglich
```

Namenskonflikte

```
public interface A {
    public static final int max = 100;
    public abstract void f();
}
```

```
public interface B {
    public static final int max = 200;
    public abstract void f();
}
```

```
public class X implements A, B {
    public void f() {
   //  A.max, B.max      eindeutig qualifizierbar
    }
}
```

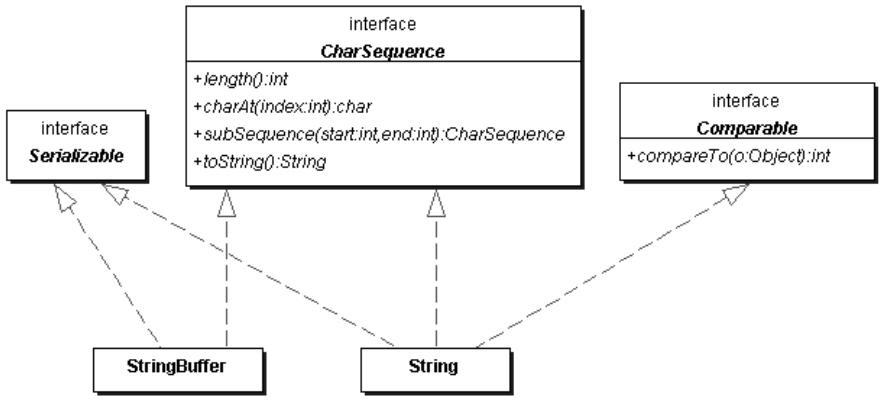

Default-Implementierungen

- Beispiel für die Anwendung von Default-Implementierungen

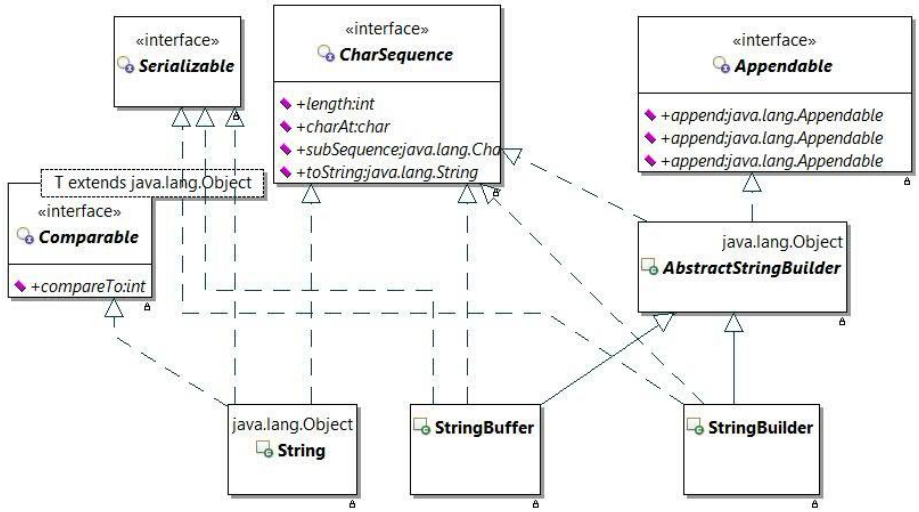
```
public interface Iterator<E> {
    boolean hasNext();

    E next();

    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
}
```

- Eine implementierende Klasse ist nicht gezwungen, die Methode `remove` zu implementieren

Beispiel: CharSequence



Beispiel: CharSequence



Die Klasse java.lang.Object (1)

Methode	*Beschreibung*
`String toString ()`	Wert des Objektes als String zurückzugeben
`boolean equals (Object o)`	Vergleich des aktuellen Objektes mit dem übergebenen `true`: wenn Objekte identisch sind muss überladen werden, falls inhaltliche Gleichheit überprüft werden soll
`int hashCode ()`	liefert einen eindeutigen Hashcode für das aktuelle Objekt, Anwendung in Hashtabellen `java.util.Hashtable`

Die Klasse java.lang.Object (2)

Methode	*Beschreibung*
`Object protected clone () throws CloneNotSupportedException`	Kopieren eines Objektes, muss aber explizit überladen werden, kann auch verboten werden `class X implements Cloneable {...}`
`@Deprecated(since="9")` `void finalize () throws Throwable`	letzte Aktion vor dem Entfernen aus dem Speicher
`Class final getClass()`	gibt ein Objekt vom Typ `Class` zurück

Beispiel: Cloneable (1)

```
public class Person implements Comparable<Person>,
                               Cloneable {
    ...

    /**
     * Kopieren des Objektes, redefiniert die
     * entsprechende Methode von Object
     */
    @Override
    public Object clone() {
        return new Person(name, vorname);
    }
    ...
```


Beispiel: Cloneable (2)

```
public class PersonTest5 {
    public void start() {
       Person p1 = new Person("Meier", "Sepp");
       Person p2 = (Person)p1.clone();
       Person p3 = new Person(p1);
       p1.setVorname("Otto");
       p3.setName("Schmidt");
       System.out.println("p1: " + p1);
       System.out.println("p2: " + p2);
       System.out.println("p3: " + p3);
    }


    public static void main(String[] args) {
        new PersonTest5().start();
    }
}
/* Ausgabe:
p1: Meier, Otto
p2: Meier, Sepp
p3: Schmidt, Sepp  */
```

Die Klasse java.lang.Class

- Für jede Klasse, jedes Interface und jeden elementaren Datentyp im System existiert ein Objekt der Klasse `java.lang.Class`, das diesen Typ beschreibt.
- Die Klasse `Class` wird zusammen mit dem Paket `java.lang.reflect`, der sogenannten Reflection-API, verwendet.
- Es gibt Methoden zum Laden von Klassen, zum Erfragen der Konstruktoren, Methoden, Attribute, usw.
- Angewendet wird die Reflection-API hauptsächlich von Software-Entwicklungswerkzeugen.

Die Klasse java.lang.Class

- Einige Methode der Klasse `Class`

Methode	*Beschreibung*
`String getName()`	Name der Klasse
`Class getSuperClass()`	Oberklasse zurückliefern
`static Class forName(String className)`	`Class`-Objekt für eine bestimmte Klasse zurückliefern
`Field[] getDeclaredFields()`	Referenzen auf die Beschreibungen der Attribute zurückgeben
`Method[] getDeclaredMethods()`	Referenzen auf die Beschreibungen der Methoden zurückgeben
`Constructor[] getDeclaredConstructors()`	Referenzen auf die Beschreibungen der Konstruktoren zurückgeben
`Object newInstance()`	Objekt der Klasse anlegen

Beispiel: ClassTest

```
public class ClassTest {
    public void test1() {
        String s1 = "Hallo";
        Person p1 = new Person("Meier", "Sepp");
        int[] tab = new int[100];
        int i = 1;
        printClassName(s1);
        printClassName(p1);
        printClassName(tab);
        printClassName(i);
    }

    public void printClassName (Object obj) {
       System.out.println ("Die Klasse von " + obj
                    + " ist " + obj.getClass().getName() );
    }
   ...
```

```
Die Klasse von Hallo ist java.lang.String
Die Klasse von Meier, Sepp ist person4.Person
Die Klasse von [I@1b90be ist [I
Die Klasse von 1 ist java.lang.Integer
```


Klassenliterale

- Ein **Klassenliteral** ist ein Ausdruck, der eine Referenz auf ein Klassenobjekt erzeugt, das einen bestimmten Typ identifiziert.
- Syntax:

  Typangabe.**class**

Ausdruck	***Ausgabe mit System.out.println***
`Class c1 = String.class;`	`class java.lang.String`
`Class c2 = Person.class;`	`class person4.Person`
`Class c3 = Comparable.class;`	`interface java.lang.Comparable`
`Class c4 = double.class;`	`double`
`Class c5 = Double.TYPE;`	`double  // Klasse Double`
`Class c6 = int[].class;`	`class [I`
`Class c7 = int[][].class;`	`class [[I`
`Class c8 = String[].class;`	`class [Ljava.lang.String;`
`Class c9 = void.class;`	`void`

Klasse ClassInfo (1)

```
import java.lang.reflect.*;
import java.util.Scanner;
public class ClassInfo {
    private Class c;
    private Object o;

    public ClassInfo(Class c) {
        this.c = c;
    }

    public void info() {
        System.out.println("\nKlasse: " + c);
        System.out.println("\nOberklasse: " + c.getSuperclass());
        System.out.println("\nKonstruktoren:");
        infoConstructors();
        System.out.println("\nAttribute:");
        infoFields();
        System.out.println("\nMethoden:");
        infoMethods();
    }
```


Klasse ClassInfo (2)

```
    public void infoConstructors() {
        Constructor ctab[] = c.getDeclaredConstructors();
        for (Constructor constr : ctab)
            System.out.println(constr);

    }

    public void infoFields() {
        Field f[] = c.getDeclaredFields();
        for (Field field : f)
            System.out.println(field);
    }

    public void infoMethods() {
        Method m[] = c.getDeclaredMethods();
        for (Method method : m)
            System.out.println(method);
    }
```

Klasse ClassInfo (3)

```
    public void newInstance() {
        try {
            System.out.println("\nObjekt der Klasse anlegen");
            o = c.newInstance();
            System.out.println("Objektinhalt: " + o);
        } catch (Throwable e) {
            System.out.println("Ausnahme: " + e);
        }
    }
```


Klasse ClassInfo (4)

```
    /** Versuche zu einem übergebenen Klassennamen ein Class-Objekt zu
     *  erzeugen und die zugehörigen Informationen auszugeben
     *  args[0]  Klassen- oder Typname      */
    public static void main(String args[]) {
        Scanner input = new Scanner(System.in);
        String name;
        if (args.length > 0)
            name = args[0];
        else {
            System.out.print("Klassenname eingeben: ");
            name = input.next();
        }
        try {
            Class c = Class.forName(name);
            ClassInfo ci = new ClassInfo(c);
            ci.info();
            ci.newInstance();
        }
        catch (Throwable e) {
            System.err.println(e);
        }
    }
}
```