

# Kapitel 1: Datentypen

- Statische Typisierung
- Primitive Datentypen und Referenztypen
- Klassen
- Interface und dynamische Bindung
- Vererbung
- Abstrakte Klassen
- Immutable Klassen und Records
- Statische Fabrikmethoden
- Subtypen und Zuweisungsregel
- Ausblick
  - Geschachtelte Klassen und Interfaces
  - Generische Typen
  - Funktionale Interfaces und Lambda-Ausdrücke

# Statische Typisierung

---

- Java ist **statisch typisiert**.
- Jede Variable und jeder Ausdruck hat einen Typ, der bereits zur **Compilierzeit** festliegt.
- Damit ist auch festgelegt, welche Operationen (Methoden, Funktionen) auf eine Variable bzw. Ausdruck angewandt werden dürfen.

```
int x = 10;  
x = 5*x;  
x = "abc";           // Compilerfehler
```

```
String s = "abc";  
s = s + "def";  
double y = Math.sqrt(s); // Compilerfehler
```

# Als Vergleich: Python ist dynamisch typisiert!

---

- Python ist dynamisch typisiert.
- Typprüfung findet erst zur Laufzeit statt.
- Erst zur Laufzeit wird geprüft, ob eine Operation zulässig ist.

```
import math

x = 5.0
x = 2.0*x
y = math.sqrt(x)

x = "abc"
y = math.sqrt(x)      # Laufzeitfehler
```

# Datentypen in Java

---

- In Java gibt es:
  - Primitive Datentypen: int, double, ...  
(sind bereits in der Sprache eingebaut)
  - Referenztypen Felder  
(sind bereits in der Sprache eingebaut)
  - Referenztypen Klassen und Interfaces:  
String, Integer, Person, ...  
(in Java API vorhanden oder selbst programmieren)

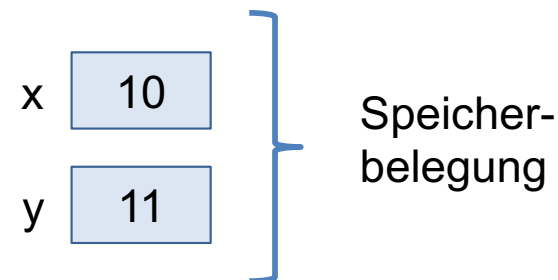
# Primitive Datentypen

- Datentypen haben festen Wertebereich (Menge von Literalen).
- Fest in die Sprache eingebaute Operationen.
- Bei Zuweisungen wird der Wert kopiert.

Primitiver Datentyp	Wertebereich	Operationen
<b>boolean</b>	<b>true, false</b>	&&,   , !, ==, !=
<b>byte, short, int, long, char</b>	..., -2, -1, 0, 1, 2, ...	+, -, *, /, %, ++, --, <, <=, ...
<b>float, double</b>	z.B. 5.3E-02	+, -, *, /, <, <=, ...

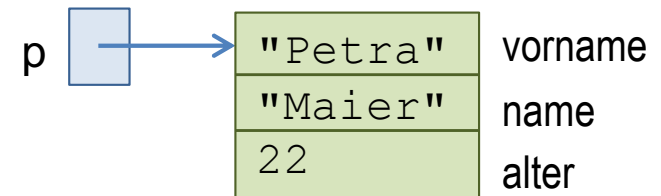
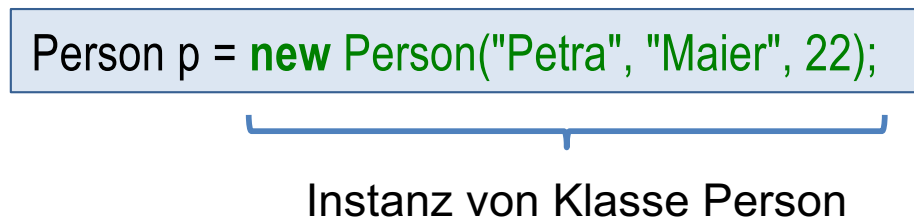
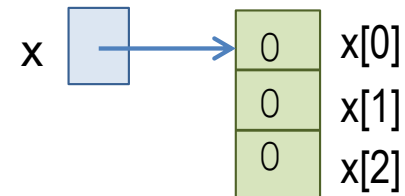
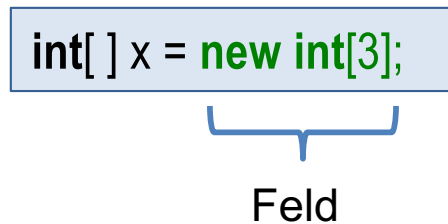
Datentyp      Variable      Wert

```
int x = 10;  
int y = x;  
y++;
```



# Referenztypen (1)

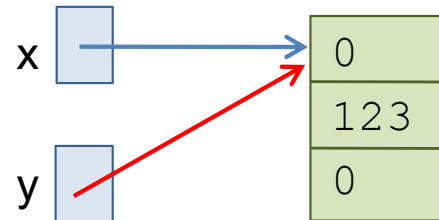
- Klassen, Interfaces und Felder.
- Haben als Werte **Referenzen** (d.h. Verweise oder Zeiger) auf Objekte.
- **Objekte** sind dynamisch mit new angelegte **Felder** oder **Instanzen von Klassen**.



# Referenztypen (2)

- Einige in die Sprache eingebaute Operationen wie: `x[i]`, `x.length` (nur für Felder), `==`, `!=`
- Ansonsten in Klassen und Interfaces selbst definierte Methoden.
- Bei Zuweisungen wird die Referenz kopiert!

```
int[] x = new int[3];  
int[] y = x;  
y[1] = 123;
```



# Kapitel 1: Datentypen

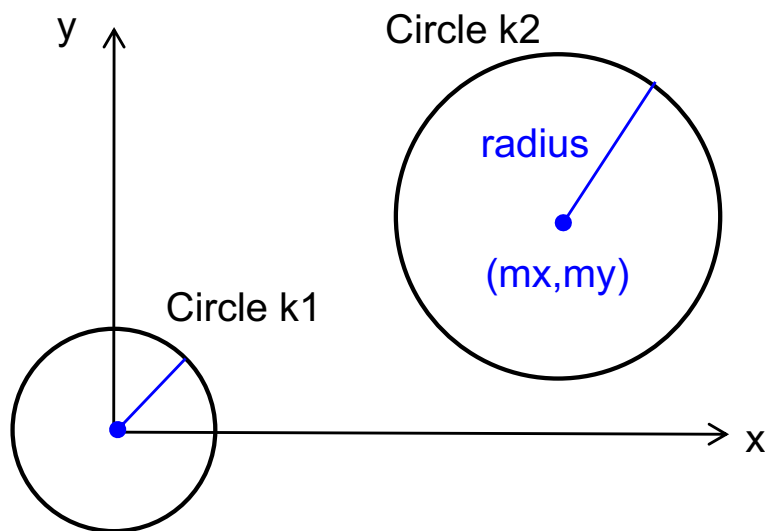
- Statische Typisierung
- Primitive Datentypen und Referenztypen
- **Klassen**
- Interface und dynamische Bindung
- Vererbung
- Abstrakte Klassen
- Immutable Klassen und Records
- Statische Fabrikmethoden
- Subtypen und Zuweisungsregel
- Ausblick
  - Geschachtelte Klassen und Interfaces
  - Generische Typen
  - Funktionale Interfaces und Lambda-Ausdrücke



# Klassen

- Eine Klasse ist ein **Bauplan für Objekte** (Instanzen) mit gleichen Eigenschaften.
- Eine Klasse legt **Instanzvariablen** und **Methoden** fest.

Circle-Objekte



Klasse Circle

```
public class Circle {  
    private double mx = 0;  
    private double my = 0;  
    private double radius = 1;  
    public void move(double x, double y) {...}  
    public void setRadius(double r) { ...}  
    public double area() { ... }  
    // ...  
}
```

Instanz-  
variablen

Methoden

# Klasse Circle

```
public class Circle {  
    public final static double PI = 3.14;  
    private double mx = 0;  
    private double my = 0;  
    private double radius = 1;  
  
    public Circle() { }  
    public Circle(double x, double y, double r) {  
        mx = x;  
        my = y;  
        radius = r;  
    }  
  
    public void move(double x, double y) {  
        mx += x;  
        my += y;  
    }  
  
    public void setRadius(double r) {  
        radius = r;  
    }  
  
    public double area() {  
        return PI * radius * radius;  
    }  
}
```

konstante (final)  
Klassenvariable (static)

Instanzvariablen

Konstruktoren

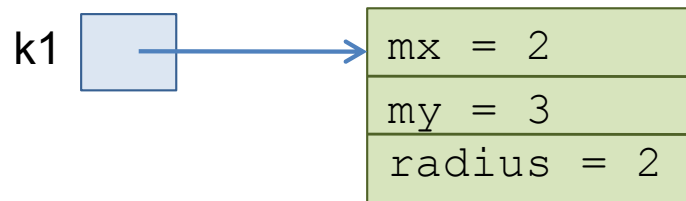
Methoden

# Benutzung der Klasse Circle

```
public class CircleApplication {  
    public static void main(String[] args) {  
        Circle k1 = new Circle();  
        k1.move(2,3);  
        k1.setRadius(2);  
        System.out.println(k1.area());    // (1)  
        // Ausgabe: PI*2*2 = 12.56  
  
        Circle k2 = k1;  
        k2.setRadius(4);  
        System.out.println(k1.area());    // (2)  
        // Ausgabe: PI*4*4 = 50.24  
    }  
}
```

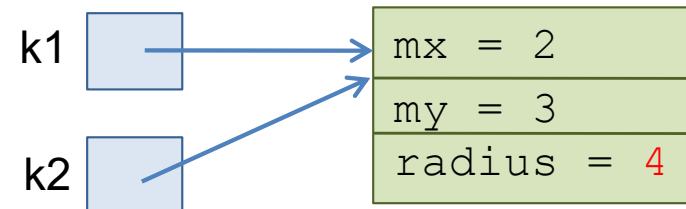
Anwendungs-  
programm

Speicherbelegung bei (1):



Kreis-Objekt

Speicherbelegung bei (2):



# Klasse = Typ + Implementierung

---

```
public class Circle {  
    ...  
    public void move(double x, double y) {...}  
    public void setRadius(double r) {...}  
    public double area() {...}  
}
```

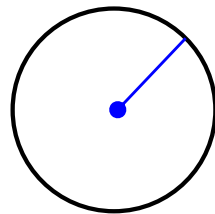
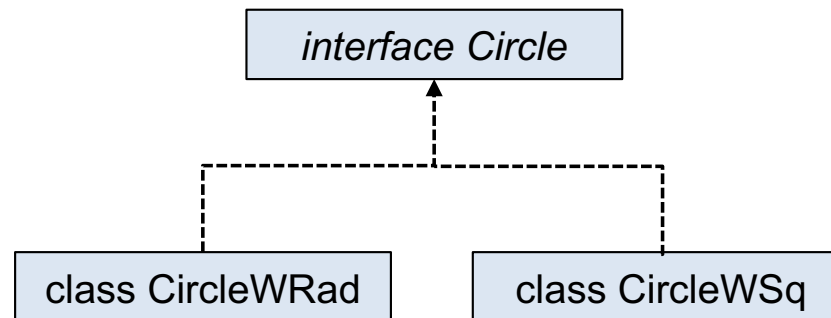
- Die Klasse Circle definiert einen **Typ Circle**.
- Jede Variable vom Typ Circle lässt damit den Aufruf der public-Methoden `move`, `setRadius` und `area` zu.
- Die Klasse Circle legt aber auch die **Implementierung** fest:
  - Instanzvariablen und
  - Definition der Methoden.

# Kapitel 1: Datentypen

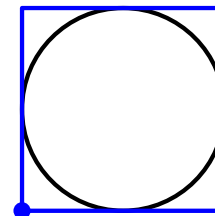
- Statische Typisierung
- Primitive Datentypen und Referenztypen
- Klassen
- **Interface und dynamische Bindung**
- Vererbung
- Abstrakte Klassen
- Immutable Klassen und Records
- Statische Fabrikmethoden
- Subtypen und Zuweisungsregel
- Ausblick
  - Geschachtelte Klassen und Interfaces
  - Generische Typen
  - Funktionale Interfaces und Lambda-Ausdrücke

# Interface: gemeinsamer Typ für verschiedene Implementierungen

- Es sind Circle-Klassen mit verschiedenen Implementierungen aber mit der gleichen Schnittstelle (interface) gewünscht.
- Implementierung soll zur Laufzeit ausgewählt werden können.



Circle mit  
Mittelpunkt und  
Radius



Circle mit linken  
unteren Punkt  
und Quadrat

# Interfaces in Java

---

- entspricht einer Klasse ohne Instanzvariablen und Konstruktor.
- es sind keine Instanzen (d.h. kein new-Aufruf ) eines Interface möglich.
- Methoden dürfen vordefiniert werden: **default-Methoden**. Sie werden dazu mit dem Schlüsselwort **default** gekennzeichnet.
- Methoden, die keine default-Implementierung haben, werden auch **abstrakt** genannt.
- es dürfen auch **statische Methoden** definiert werden.
- **Konstante** sind erlaubt (sind implizit final und static)

# Beispiel: Interface Circle

---

```
public interface Circle {  
    double PI = 3.14;  
    void move(double x, double y);  
    void setRadius(double r);  
    double area();  
}
```

Interface Circle  
nur mit abstrakten  
Methoden

```
public interface Circle {  
    double PI = 3.14;  
    void move(double x, double y);  
    void setRadius(double r);  
    double getRadius();  
  
    default double area() {  
        double r = getRadius();  
        return PI * r * r;  
    }  
}
```

Interface Circle mit  
einer default-Methode



# Beispiel: Interface mit statischen Methoden

---

```
interface Formelsammlung {  
    static double kugelVolumen(double radius) {  
        return (4.0/3.0) * Math.PI * Math.power(radius,3);  
    }  
  
    // ...  
}
```

# Implementierung eines Interface

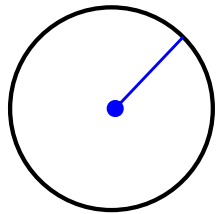
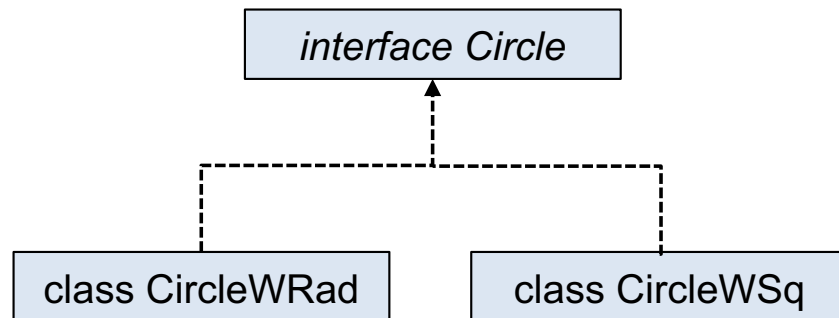
---

- Eine Klasse **implementiert** ein Interface, in dem alle abstrakten Methoden des Interface definiert werden.

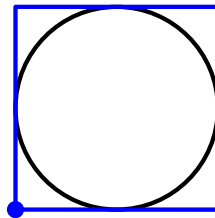
```
public interface BeispielInterface {  
    B f(A1 a1, ...);  
    // ...  
}
```

```
public class BeispielKlasse implements BeispielInterface {  
    // Daten:  
    // ...  
  
    public B f(A1 a1, ...) {...}  
  
    // Weitere Methoden:  
    // ...  
}
```

# Interface Circle mit versch. Implementierungen (1)



Circle mit  
Mittelpunkt und  
Radius



Circle mit linken  
unteren Punkt  
und Quadrat

```
public interface Circle {
    double PI = 3.14;
    void move(double x, double y);
    void setRadius(double r);
    double area();
}
```

```
public class CircleWRad
implements Circle {
    //...
}
```

```
public class CircleWSq
implements Circle {
    //...
}
```

# Interface Circle mit versch. Implementierungen (2)

---

```
public class CircleWRad implements Circle {
    private double mx = 0;
    private double my = 0;
    private double radius = 1;

    public CircleWRad() { }

    public CircleWRad(double x, double y, double r) {
        mx = x;
        my = y;
        radius = r;
    }

    public void move(double x, double y) {
        mx += x;
        my += y;
    }

    public void setRadius(double r) {
        radius = r;
    }

    public double area() {
        return PI* radius * radius;
    }
}
```

# Interface Circle mit versch. Implementierungen (3)

---

```
public class CircleWSq implements Circle {
    private double llX = -1; // x-Koord. des linken unteren Punkts
    private double llY = -1; // y-Koord. des linken unteren Punkts
    private double len = 2;   // Kantenlaenge des Quadrats

    public CircleWSq() { }

    public CircleWSq(double x, double y, double r) {
        llX = x-r;
        llY = y-r;
        len = 2*r;
    }

    public void move(double x, double y) {
        llX += x;
        llY += y;
    }

    public void setRadius(double r) {
        len = 2*r;
    }

    public double area() {
        return PI * len * len / 4;
    }
}
```

# Interface Circle mit versch. Implementierungen (4)

```
public class CircleApplication {  
  
    private CircleApplication() { }  
  
    public static void main(final String[] args) {  
        Circle k;  
        if (...)  
            k = new CircleWRad(1,1,3);  
        else  
            k = new CircleWSq(1,1,3);  
  
        System.out.println(k.area());  
    }  
}
```

Auswahl der  
Implementierung  
zur Laufzeit.

## Dynamische Bindung:

Erst zur Laufzeit (dynamisch) wird der Methodenaufruf an die Definition der Methode gebunden.

Welche Methode gebunden wird, hängt davon ab, von welcher Klasse das Objekt ist, auf das k verweist.

# Dynamische Bindung

---

- Ein Methodenaufruf

`x.fun(y);`

wird erst zur Laufzeit (daher dynamisch) an eine Methodendefinition

```
class A ... {  
    public ... fun(B b) {...}  
    ...  
}
```

gebunden.

- Dabei ist die Klasse `A` im allgemeinen nicht der deklarierte Typ von `x` sondern die Klasse des Objekts, auf das `x` verweist.
- Der deklarierte Typ von `x` wird auch statischer Typ genannt (zur Übersetzungszeit bekannt). Dagegen wird der Typ des Objekts, auf den `x` verweist, auch dynamischer Typ genannt (erst zur Laufzeit bekannt). Dann bedeutet dynamische Bindung die Bindung an den dynamischen Typ.
- Anmerkung: in C++ muss die dynamische Bindung durch das Schlüsselwort `virtual` erzwungen werden.

# Beispiel

- Statische Methode `move`, die eine Menge von Kreisen um `x`, `y` verschiebt.
- Die Kreise werden als Feld übergeben.

```
public class CircleApplication {  
  
    public static void move(Circle[] circleArray, double x, double y) {  
        for (Circle k : circleArray)  
            k.move(x, y);  
    }  
  
    public static void main(final String[] args) {  
        Circle[] circleArray = new Circle[3];  
        circleArray[0] = new CircleWRad(1,1,3);  
        circleArray[1] = new CircleWSq(0,1,3);  
        circleArray[2] = new CircleWSq(1,0,3);  
  
        move(circleArray , 3, 4);  
    }  
}
```



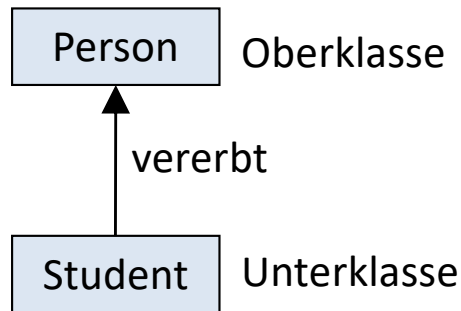
# Kapitel 1: Datentypen

- Statische Typisierung
- Primitive Datentypen und Referenztypen
- Klassen
- Interface und dynamische Bindung
- Vererbung
- Abstrakte Klassen
- Immutable Klassen und Records
- Statische Fabrikmethoden
- Subtypen und Zuweisungsregel
- Ausblick
  - Geschachtelte Klassen und Interfaces
  - Generische Typen
  - Funktionale Interfaces und Lambda-Ausdrücke

# Vererbung (1)

---

- Mit **Vererbung (inheritance)** lässt sich eine gegebene Klasse A in eine neue Klasse B um zusätzliche Eigenschaften erweitern.  
Die Klasse A wird auch als **Oberklasse** und die Klasse B als **Unterklasse** bezeichnet.
- Mit Vererbung lässt sich auch ein Interface IA in ein Interface IB erweitern.



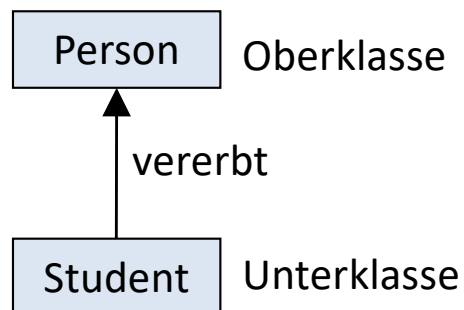
```
public class Person {  
    // ...  
}
```

```
public class Student extends Person {  
    // ...  
}
```

# Vererbung (2)

Um eine gegebene Klasse A in eine neue Klasse B zu erweitern, gibt es verschiedene Möglichkeiten:

- B kann um **zusätzliche Instanzvariablen** erweitert werden.
- B kann um **zusätzliche Methoden** erweitert werden.
- Vererbte **Methoden** aus A können **überschrieben (overriding)** werden.



```
public class Person {
    // Instanzvariablen von Person:
    ...
    // Methoden von Person:
    ...
}
```

```
public class Student extends Person {
    // Instanzvariablen von Student:
    ...
    // Überschriebene Methoden von Person:
    ...
    // Zusätzliche Methoden von Student:
    ...
}
```

# Beispiel (1)

```
public class Person {  
  
    protected String name;  
    protected int gebJahr;  
  
    public Person(String n, int j){  
        name = n;  
        gebJahr = j;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getGebJahr() {  
        return gebJahr;  
    }  
  
    public void print() {  
        System.out.print(name  
            + "; " + gebJahr);  
    }  
}
```

```
public class Student extends Person {  
  
    private String hochschule;  
  
    public Student(String n,  
                    int j, String hs) {  
        super(n, j);  
        hochschule = hs;  
    }  
  
    public String getHochschule() {  
        return hochschule;  
    }  
  
    public void print() {  
        super.print();  
        System.out.print(  
            "; immatr an " + hochschule);  
    }  
}
```

Neue  
Instanzvariable

Neue Methode

Überschriebene  
Methode

# Beispiel (2)

```
public class Anwendung {  
  
    public static void main(String[] args){  
        Person p1 = new Person("Hans",1990);  
        Student s1 = new Student("Maria",1991,"HTWG KN");  
        Person p2 = new Student("Paul",1988,"HTWG KN");  
  
        System.out.println(p1.getName());  
        System.out.println(s1.getName());  
  
        p1.print(); System.out.println();  
        s1.print(); System.out.println();  
        p2.print(); System.out.println();  
    }  
}
```

Hans  
Maria

Hans; 1990  
Maria; 1991; immatr an HTWG KN  
Paul; 1988; immatr an HTWG KN

- **Methodenaufrufe** werden **dynamisch gebunden**.  
Daher werden die print-Aufrufe wie folgt gebunden:
  - p1 verweist auf Person, daher Bindung an Person.print().
  - p2 verweist auf Student, daher Bindung an Student.print().
  - s1 verweist auf Student, daher Bindung an Student.print().
- Student darf an Person zugewiesen werden: p2 = new Student(...).  
(**Zuweisungsregel**; kommt später)

# Überschreiben einer Methode (Overriding)

---

- Methodenname bleibt gleich
- gleiche Anzahl Parameter
- gleiche Typen der Parameter
- Rückgabewerttyp darf spezieller werden: Rückgabewerttyp ist kovariant
- Zugriffsrechte dürfen erweitert werden.
- Annotation `@Override` verwenden: Compiler überprüft dann, ob korrekt überschrieben wurde.  
Mit Annotation `@Override` wird Verwechslungsgefahr mit Overloading vermieden.

```
public class Person {  
    // ...  
  
    public void print() {  
        System.out.print(name  
            + "; " + gebJahr);  
    }  
}
```

```
public class Student extends Person {  
    // ...  
  
    @Override  
    public void print() {  
        super.print();  
        System.out.print(  
            "; immatr an " + hochschule);  
    }  
}
```

# Überschreiben einer Methode verbieten

- Überschreiben kann durch **final** explizit verboten werden. Damit kann eine Methode durch Vererbung nicht mehr verändert werden. (siehe auch immutable Klassen)

```
public class Shape {  
  
    // linker unterer Bezugspunkt:  
    private double x = 0;  
    private double y = 0;  
  
    public final void move(double x, double y) {  
        this.x += x;  
        this.y += y;  
    }  
}
```

```
public class Circle extends Shape {  
  
    private double radius = 1;  
  
    // ...  
}
```

Methode `move` darf  
nicht mehr über-  
schrieben werden.

# Überladen einer Methode (Overloading)

- Definition einer neuen Methode mit Namen einer bereits vorhandenen Methode.
- Jedoch: Anzahl Parameter und/oder Typen der Parameter sind unterschiedlich

```
public class Student extends Person {  
    // ...  
  
    @Override  
    public void print() {  
        super.print();  
        System.out.print(  
            "; immatr an " + hochschule);  
    }  
  
    public void print(int lz) {  
        for (int i = 0; i < lz; i++)  
            System.out.println();  
        print();  
    }  
}
```

Überladene  
print-Methode

```
public class Anwendung {  
  
    public static void main(...) {  
        Student s =  
            new Student(...);  
        Person p = s;  
  
        s.print();  
        p.print();  
        s.print(3);  
    }  
}
```

Beachte:  
p.print(3) wäre  
nicht erlaubt!



# Klasse Object (1)

---

- Die Klasse `Object` ist Oberklasse jeder anderen Klasse.
- Eine Klasse, die nicht explizit durch Vererbung (d.h. mit `extends`) definiert ist, wird implizit von `Object` vererbt.

```
public class A {  
    // ...  
}
```

ist äquivalent zu:

```
public class A extends Object {  
    // ...  
}
```

- Die Klasse `Object` ist damit die Wurzelklasse der über Vererbung definierten Vererbungshierarchie.

# Klasse Object (2)

---

Die Klasse `Object` enthält u.a. folgende Methoden:

- `public boolean equals(Object obj)`

Prüft, ob `this` und `obj` gleich sind. Die voreingestellte Implementierung prüft, ob die Referenzen gleich sind (d.h. `this == obj`).

Ist eine inhaltliche Gleichheit gewünscht, muss die Methode geeignet überschrieben werden. Dabei ist zu beachten, dass `equals` reflexiv, symmetrisch und transitiv ist.

- `protected Object clone()`

Erstellt eine Kopie des Objekts. Die voreingestellte Implementierung sieht eine flache Kopie vor. Eine flache Kopie kann gefährlich sein. Daher ist die Methode auch als `protected` definiert. Eine tiefe Kopie muss selbst programmiert werden.

- `public String toString()`

Liefert eine String-Darstellung des Objekts. Die voreingestellte Implementierung liefert den Klassen-Namen und eine Hash-Code. Jede Klasse sollte diese Methode überschreiben.

- `public int hashCode()`

Liefert einen Hash-Code (int-Wert) für das Objekt. Für unterschiedliche Objekte ist der Hash-Code meistens unterschiedlich. Für gleiche Objekte (bzgl. `equals`) muss der Hash-Code identisch sein.

# Klasse Point mit equals und toString

```
public class Point {  
  
    private final double x = 0;  
    private final double y = 0;  
  
    // ...  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o)  
            return true;  
        if (! (o instanceof Point))  
            return false;  
        Point p = (Point) o;  
        return (this.x == p.x && this.y == p.y);  
    }  
  
    @Override  
    public int hashCode() { ... }  
  
    @Override  
    public String toString() {  
        return "(" + this.x + "," + this.y + ")";  
    }  
}
```

Testet, ob o vom  
Typ Point ist.

# Empfehlung für die Definition eigener Klassen

---

- `toString` sollte für jede Klasse überschrieben werden.
- Bei Wertklassen wie z.B. `Point` sollte `equals` und `hashCode` zwingend überschrieben werden.
- Annotation `@Override` verwenden.
- Es sind unbedingt die in der *Java API Specification* festgelegten Bedingungen einzuhalten.
- Es gibt bei `equals` viele Probleme im Zusammenhang mit Vererbung.
- Eine sehr gute Darstellung der Probleme und Lösungen findet man auch in Kap. 3 in *Effective Java* von J. Bloch.

# Kapitel 1: Datentypen

- Statische Typisierung
- Primitive Datentypen und Referenztypen
- Klassen
- Interface und dynamische Bindung
- Vererbung
- **Abstrakte Klassen**
- Immutable Klassen und Records
- Statische Fabrikmethoden
- Subtypen und Zuweisungsregel
- Ausblick
  - Geschachtelte Klassen und Interfaces
  - Generische Typen
  - Funktionale Interfaces und Lambda-Ausdrücke

# Abstrakte Klassen

- Eine **abstrakte Methode** ist eine Methode ohne Implementierung. Kennzeichnung mit dem Schlüsselwort `abstract`.
- Eine **Klasse** heißt **abstrakt**, wenn sie wenigstens eine abstrakte Methode enthält. Sie wird ebenfalls mit dem Schlüsselwort `abstract` versehen.
- Von einer abstrakten Klasse können keine Instanzen erzeugt werden.
- In einer abgeleiteten Klasse können eine oder mehrere abstrakte Methoden implementiert werden.

```
public abstract class Shape{  
    private double x = 0;  
    private double y = 0;  
  
    // ...  
  
    public void move(double x, double y){  
        this.x += x;  
        this.y += y;  
    }  
  
    public abstract double getArea();  
}
```

abstrakte  
Methode

```
public class Circle extends Shape{  
    private double radius = 1;  
  
    // ...  
  
    public double getArea() {  
        return Math.PI*radius*radius;  
    }  
}
```

Implementierung der  
abstrakten Methode

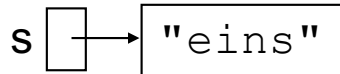
# Kapitel 1: Datentypen

- Statische Typisierung
- Primitive Datentypen und Referenztypen
- Klassen
- Interface und dynamische Bindung
- Vererbung
- Abstrakte Klassen
- Immutable Klassen und Records
- Statische Fabrikmethoden
- Subtypen und Zuweisungsregel
- Ausblick
  - Geschachtelte Klassen und Interfaces
  - Generische Typen
  - Funktionale Interfaces und Lambda-Ausdrücke

# Immutable Klassen

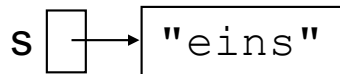
- `immutabel` = unveränderbar
- Wert eines Objekts ändert sich nicht mehr, sobald es konstruiert wurde.
- Beispiel: Klasse `String` ist eine immutable Klasse.

```
String s = "eins";
```



```
s.method(...);
```

```
fun(s);
```

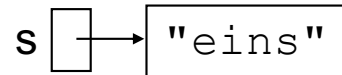


Der String "eins" kann durch Methoden-Aufrufe nicht verändert werden.



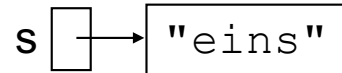
# Vor- und Nachteile

```
String s = "eins";
```



```
s.method(...);
```

```
fun(s);
```



Vorteile	Nachteile
Einfacher zu benutzen.	können wesentlich ineffizienter sein (vergleiche z.B. String und StringBuilder).
Weniger Fehlerquellen.	
Keine inkonsistenten Zustände bei nebenläufigem Zugriff (Threads, parallele Ströme)	

# Vorgehen

---

- (1) Instanzvariablen als `final` definieren.  
Instanzvariablen sind dann unveränderbar.
- (2) Vererbung verbieten durch `final class`.  
Damit kann die Klasse nicht um mutable Anteile erweitert werden.
- (3) Achtung: falls Instanzvariable x eine Referenz auf ein mutables Objekt oder ein Feld ist, dann genügt `final` nicht.  
Daher zusätzliche Maßnahmen:
  - x sollte `private` sein. Kein direkter Zugriff von außen.
  - x darf in keiner Methode verändert werden.  
Also **keine setter-Methoden!**
  - falls in einem Konstruktor mutable Objekte bzw. Felder als Parameter zum Initialisieren von x übergeben werden, dann muss x mit einer **tiefen Kopie** (defensive copy) initialisiert werden.
  - x darf nicht direkt als Rückgabewert zurückgeliefert sondern nur eine tiefe Kopie von x.

# Beispiel: Immutable Klasse für komplexe Zahlen (1)

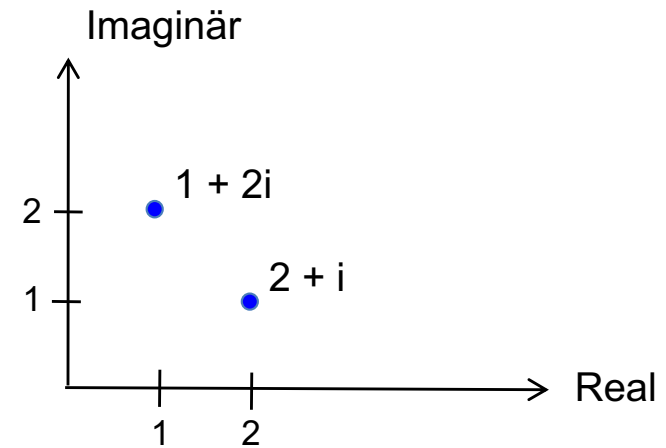
## Komplexe Zahlen:

- Zahlen der Bauart:

$$x + iy$$

wobei  $x$  und  $y$  reelle Zahlen sind  
und  $i^2 = -1$  definiert wird.

$x$  heisst auch **Realteil** und  $y$  **Imaginärteil**.



- Addition:

$$(x_1 + iy_1) + (x_2 + iy_2) = (x_1 + x_2) + i(y_1 + y_2)$$

- Multiplikation:

$$(x_1 + iy_1) * (x_2 + iy_2) = (x_1 x_2 - y_1 y_2) + i(y_1 x_2 + x_1 y_2)$$

# Beispiel: Immutable Klasse für komplexe Zahlen (2)

```
public final class Complex {  
  
    private final double re;  
    private final double im;  
  
    public Complex(double re, double im) {  
        this.re = re;  
        this.im = im;  
    }  
  
    public Complex() {  
        this(0, 0);  
    }  
  
    public Complex add(Complex v) {  
        double real = this.re + v.re;  
        double imag = this.im + v.im;  
        return new Complex(real, imag);  
    }  
  
    // ...  
}
```

Beachte:  
Zugriff auf Instanzvariablen  
von v gestattet, da v zur  
Klasse Complex gehört.

# Beispiel: Immutable Klasse für komplexe Zahlen (3)

```
public class Complex {

    // ...

    public Complex mult(Complex v) {
        double real = this.re*v.re - this.im*v.im;
        double imag = this.im*v.re + this.re*v.im;
        return new Complex(real,imag);
    }

    @Override
    public String toString() {
        return this.re + " + " + this.im + "*i";
    }

    public static void main(String[] args) {
        Complex z1 = new Complex(1,1);           // z1 = 1 + i
        Complex z2 = new Complex(1,1);           // z2 = 1 + i
        z1 = z1.mult(z1).add(z2);                 // z1 = z1*z1 + z2;
        System.out.println(z1);                  // 1.0 + 3.0*i
        z1 = z1.mult(z1).add(z2);                 // z1 = z1*z1 + z2;
        System.out.println(z1);                  // -7.0 + 7.0*i
    }
}
```

Impliziter Aufruf  
von toString().

# Beispiel: immutable Klasse Vektor mit Feld als Instanzvariable

```
public final class Vector {  
    private final double[] coords;  
  
    public Vector(double[] a) {  
        coords = new double[a.length];  
        System.arraycopy(a, 0, coords, 0, a.length);  
    }  
  
    public Vector plus(Vector x) {  
        assert this.coords.length == x.coords.length;  
        Vector z = new Vector(coords);  
        for (int i = 0; i < coords.length; i++)  
            z.coords[i] += x.coords[i];  
        return z;  
    }  
  
    public void set(int i, double x) {  
        coords[i] = x;  
    }  
  
    public double[] get() {  
        return coords;  
    }  
}
```

Instanzvariable ist ein Feld!

Feld als Parameter zum Initialisieren der Instanzvariable.  
Daher: tiefe Kopie!

Methode set ist nicht erlaubt,  
da set(i,x) den Wert von  
coords[i] verändern würde!

Keine direkte Rückgabe der  
Instanzvariable.  
Rückgabe einer tiefen Kopie  
wäre gestattet.

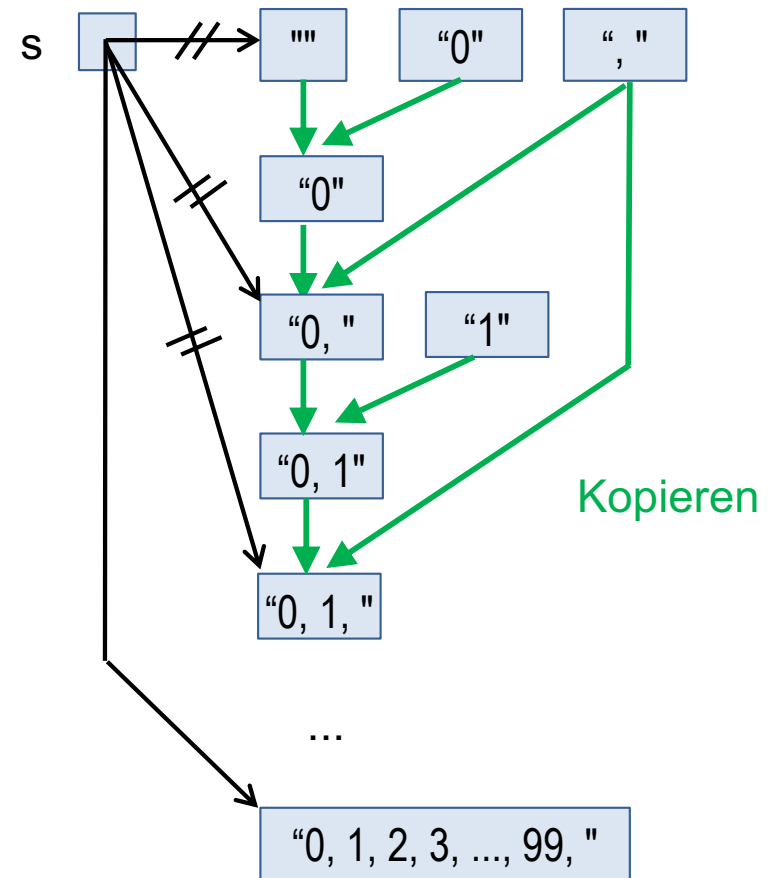
# Ineffizienz bei immutabler Klasse String

```
String s = "";
```

```
for (int i = 0; i < 100; i++)  
    s = s + i + ", ";
```

```
System.out.println(s); // "0, 1, ..., 99, "
```

- großer Kopieraufwand
- viel Heap-Speicher



# Mutable Klasse StringBuilder

---

- StringBuilder und StringBuffer (thread-sichere-Variante) sind mutable Varianten der Klasse String.
- Die Zeichenfolgen können verändert werden.
- Wenn ein String aus vielen Teilstücken zusammengesetzt (konkatinert) werden muss, sind mit StringBuilder und StringBuffer wesentlich **effizientere Lösungen** möglich.

```
StringBuilder s = new StringBuilder("");  
  
for (int i = 0; i < 100; i++)  
    s.append(i).append(", ");  
  
System.out.println(s);
```

- s.append(t) hängt den String t an den String(Builder) s an.
- Da append eine Referenz auf das Objekt zurückliefert, lässt sich append mehrfach anwenden (verketteten):

s.append(t1).append(t2)...



# Records als Kurzform für immutable Klassen

```
public final class Complex {  
  
    private final double real;  
    private final double im;  
  
    public Complex(double re, double im) {  
        this.real = re;  
        this.im = im;  
    }  
  
    public im() { return this.im; }  
  
    public real() { return this.real; }  
  
    @Override  
    public boolean equals(Object o) { ... }  
  
    @Override  
    public int hashCode() { ... }  
  
    @Override  
    public String toString() { ... }  
  
    // weitere Methoden: ...  
}
```



gleichwertig aber  
wesentlich kürzer!

```
public record Complex(double re, double im) {  
  
    // weitere Methoden: ...  
}
```

- Beachte:  
records sind implizit final  
(also keine Vererbung)
- Seit Java 14.

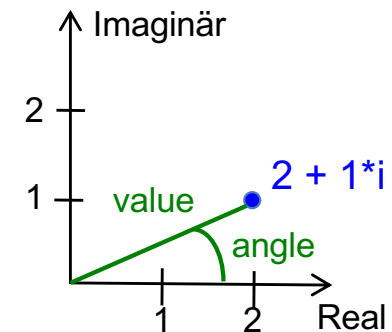
# Kapitel 1: Datentypen

- Statische Typisierung
- Primitive Datentypen und Referenztypen
- Klassen
- Interface und dynamische Bindung
- Vererbung
- Abstrakte Klassen
- Immutable Klassen und Records
- Statische Fabrikmethoden
- Subtypen und Zuweisungsregel
- Ausblick
  - Geschachtelte Klassen und Interfaces
  - Generische Typen
  - Funktionale Interfaces und Lambda-Ausdrücke

# Statische Fabrikmethode

- Statische Fabrikmethoden sind eine weitere (oft vorteilhafte) Alternative zur Erzeugung von Instanzen.
- Beispiel: Klasse Complex

```
public final class Complex {  
    private final double re = 0;  
    private final double im = 0;  
    private Complex(double re, double im) {  
        this.re = re;  
        this.im = im;  
    }  
    public static Complex newComplexFromPolar(double angle, double value) {  
        return new Complex(value*Math.cos(angle), value*Math.sin(angle));  
    }  
    public static Complex newComplexFromCartesian(double re, double im) {  
        return new Complex(re, im);  
    }  
    public void main( ... ) {  
        Complex z1 = newComplexFromPolar(Math.PI/6, Math.sqrt(5));  
        Complex z2 = newComplexFromCartesian(2, 1);  
    }  
}
```



Statische  
Fabrikmethoden zum  
Erzeugen von komplexen  
Zahlen.

Durch Einschränkung  
des Konstruktors auf  
privaten Zugriff ist das  
Erzeugen von komplexen  
Zahlen nur über die  
Fabrikmethoden möglich.

# Bemerkungen

---

- Statische Fabrikmethoden bieten mehr Möglichkeiten.
- Z.B. lassen sich die Fabrikmethoden `newComplexFromPolar` und `newComplexFromCartesian` nicht gleichwertig durch Konstruktoren ersetzen.
- Fabrikmethoden müssen nicht unbedingt Objekte neu erzeugen, sondern können auch auf bereits vorgefertigte Objekte zurückgreifen ([Cache-Mechanismus](#)).

```
public final class Complex {  
    // ...  
    private static Complex zero;  
    private static Complex one;  
    public static Complex newComplexFromReal(double x) {  
        if (x == 0) {  
            if (zero == null) zero = new Complex(0,0);  
            return zero;  
        } else if (x == 1) {  
            if (one == null) one = new Complex(1,0);  
            return one;  
        } else {  
            return new Complex(x, 0);  
        }  
    }  
}
```

# Kapitel 1: Datentypen

- Statische Typisierung
- Primitive Datentypen und Referenztypen
- Klassen
- Interface und dynamische Bindung
- Vererbung
- Abstrakte Klassen
- Immutable Klassen und Records
- Statische Fabrikmethoden
- Subtypen und Zuweisungsregel
- Ausblick
  - Geschachtelte Klassen und Interfaces
  - Generische Typen
  - Funktionale Interfaces und Lambda-Ausdrücke

# Subtyp-Beziehung

---

- Jede Klasse und jedes Interface definiert einen Typ mit gleichem Namen.
- Durch jede Vererbungs- und Implementierungsbeziehung wird eine Subtyp-Beziehung (Untertypbeziehungen) definiert.

```
class A extends B {  
    // ...  
}
```

```
interface A extends B {  
    // ...  
}
```

```
class A implements B {  
    // ...  
}
```

A ist Subtyp (Untertyp) von B

B  
↑  
A

A <: B

# Beispiel mit Subtypen

- Klassen und Interfaces, für die keine Vererbungsbeziehung definiert sind, sind implizit Subtyp von Object.

```
interface Shape {  
    double getArea();  
    // ...  
}
```

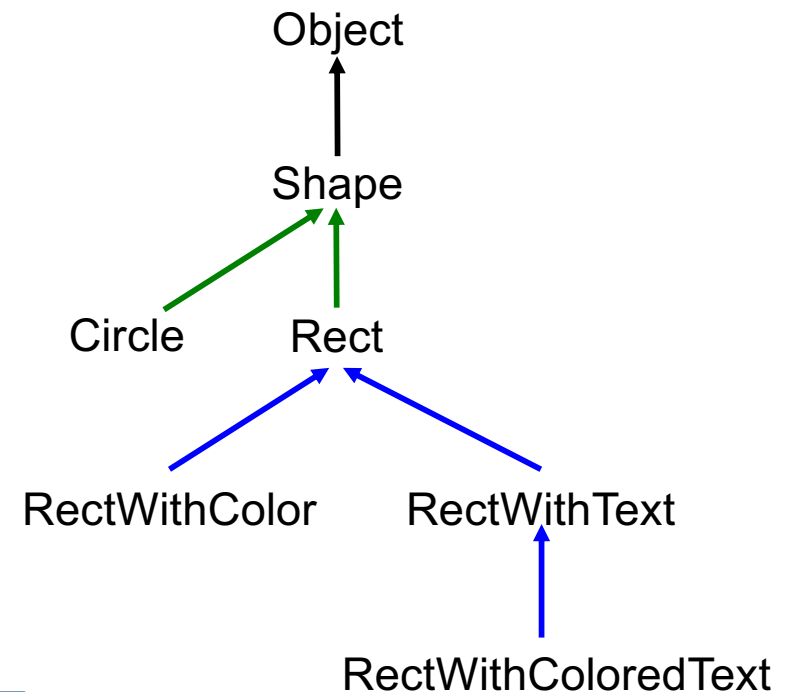
```
class Circle implements Shape { ... }
```

```
class Rect implements Shape { ... }
```

```
class RectWithColor extends Rect { ... }
```

```
class RectWithText extends Rect { ... }
```

```
class RectWithColoredText extends RectWithText { .... }
```



Beziehungen, die sich durch Reflexivität und Transitivität ergeben, sind weggelassen.

# Zuweisungsregel

---

- Einer Variablen dürfen nur Werte von Subtypen zugewiesen werden.  
Das betrifft Zuweisungen, Parameterübergabe und Rückgabe.

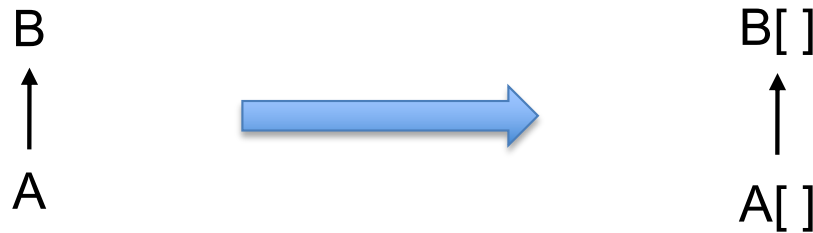
```
Rect r = new RectWithColor();  
Shape s = new Circle();  
s = r;  
r = s; // Typfehler !!!
```

```
class Plane {  
    // ...  
    void add(Shape s) {...}  
  
    public static void main(...) {  
        Plane p = new Plane();  
        p.add(new Circle());  
        p.add(new Rect());  
        p.add(new RectWithColor());  
    }  
}
```

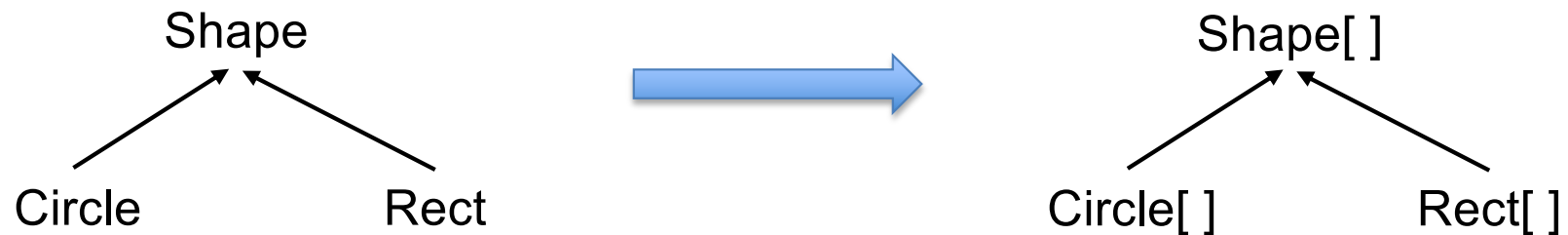


# Felder sind kovariant

- Falls  $A <: B$ , dann ist  $A[] <: B[]$



- Beispiel



# Beispiel mit Kovarianz von Feldern

- Da Rect[ ] und Circle[ ] Subtypen von Shape[ ] sind, ist folgendes Programm korrekt typisiert. Geschickt!

```
class ShapeUtility {  
  
    public static double totalArea(Shape[] shapes) {  
        double total = 0;  
        for (Shape s : shapes)  
            total += s.getArea();  
        return total;  
    }  
  
    public static void main(...) {  
        Rect[] rs = new Rect[5];  
        rs[0] = new RectWithColor();  
        // ...  
        Circle[] cs = new Circle[5];  
        cs[0] = new Circle();  
        // ...  
        System.out.println("Summe der Rechteckflaechen: " + totalArea(rs);  
        System.out.println("Summe der Kreisflaechen: " + totalArea(cs);  
    }  
}
```

# Kovarianz von Feldern ist eigentlich falsch

---

- Kovarianz von Feldern kann zu **Laufzeitfehlern** führen!

```
Shape[ ] s = new Rect[5];  
s[0] = new Circle();
```

Bei beiden Zuweisungen wird die Zuweisungsregel eingehalten. Der Compiler liefert daher keinen Fehler.

**Fehler zur Laufzeit: `ArrayStoreException`.**  
s ist zur Laufzeit ein Rect-Feld und kann kein Circle-Objekt abspeichern.

- Die **kovariante Typisierungsregel für Felder ist eigentlich falsch**:  
Ein Shape-Feld ist in der Lage sowohl Kreise als auch Rechtecke zu speichern.  
Ein Rechteckfeld kann das nicht.  
Daher dürfte Rechteck-Feld kein Subtyp von Shape-Feld sein!
- Die Kovarianzregel für Felder ist eine Altlast aus früheren Java-Versionen.  
Wie wir sehen werden, sind Collections (Behälter; lassen sich alternativ zu Feldern verwenden) nicht kovariant und führen daher nicht zu Laufzeitfehlern.

# Kovarianz von Feldern in java.util.Arrays

---

- die Klasse Arrays enthält verschiedene statische Methoden, die sich die Kovarianz von Feldern zu Nutze machen.
- Beispiel:

```
// Returns a string representation of the contents of the specified array.  
public static String toString(Object[ ] a);
```

```
String[ ] sf = {"abc","def","ghi"};  
System.out.println(Arrays.toString(sf));
```

Aufgrund der Kovarianz von Feldern gilt:

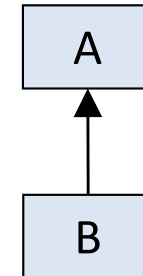
`String[ ] <: Object[ ]`

# Liskovsches Substitutionsprinzip

---

Liskovsches Substitutionsprinzip:

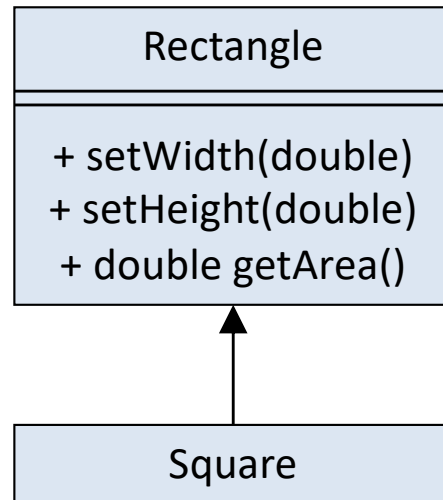
Falls B Subtyp von A ist, dann sollte in einem Programm ein A-Objekt problemlos durch ein B-Objekt ersetzt werden können.



Vererbungs- und Implementierungsbeziehungen sollten immer so definiert werden, dass das Liskovsche Substitutionsprinzip eingehalten wird.

# Verletzung des Substitutionsprinzips bei Square und Rectangle

---

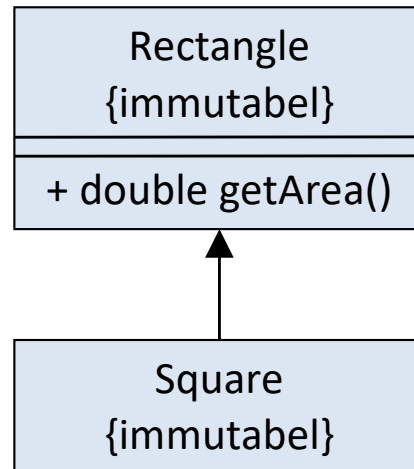


Verletzung des  
Substitutionsprinzips

- Bei einem Rechteck lässt sich die Breite unabhängig von der Höhe verändern. Wenn in einem Programmstück die Breite eines Rechtecks verdoppelt wird, dann verdoppelt sich auch die Fläche.
- Das gilt nicht für ein Quadrat. Bei Verdopplung der Breite vervierfacht sich die Fläche!
- Daher verstößt die obere Vererbungsbeziehung gegen das Liskovsche Substitutionsprinzip.

# Keine Verletzung des Substitutionsprinzips bei immutablen Klassen

---



- Ein immutables Rechteck lässt sich problemlos durch ein immutables Quadrat ersetzen.
- Wie in der Geometrie:  $\text{Square} \subseteq \text{Rectangle}$

# Kovarianz von Feldern verletzt Substitutionsprinzip

- Felder sind in Java kovariant:



- Ersetzbarkeit ist nicht gegeben.  
Daher Verstoß gegen das Liskovsche Substitutionsprinzip!

```
Shape[] s = new Rect[5];  
s[0] = new Circle();
```

Fehler zur Laufzeit: `ArrayStoreException`.  
s ist zur Laufzeit ein `Rect`-Feld und kann kein `Circle`-Objekt abspeichern.



# Kapitel 1: Datentypen

- Statische Typisierung
- Primitive Datentypen und Referenztypen
- Klassen
- Interface und dynamische Bindung
- Vererbung
- Abstrakte Klassen
- Immutable Klassen und Records
- Statische Fabrikmethoden
- Subtypen und Zuweisungsregel
- Ausblick
  - Geschachtelte Klassen und Interfaces
  - Generische Typen
  - Funktionale Interfaces und Lambda-Ausdrücke

# Geschachtelte Klassen und Interfaces

---

- Klassen (und auch Interfaces) können geschachtelt sein.
- Damit lassen sich gewünschte Kopplungen zwischen Klassen erzielen.
- Es gibt verschiedene Ausprägungen:
  - statisch geschachtelte Klassen (static nested classes)
  - innere Klassen (inner classes)
  - lokale innere Klassen (local classes)
  - anonyme innere Klassen (anonymous classes)

# Statisch geschachtelte Klassen

---

```
class EnclosingClass {  
  
    static class StaticNestedClass{  
        ...  
    }  
  
    ...  
}
```

- **Kopplung auf Klassenebene.**
- Die statisch geschachtelte Klasse StaticNestedClass verhält sich wie eine Top-Level-Klasse (d.h. nicht-geschachtelte Klasse).
- Jedoch hat die äußere Klasse alle Zugriffsrechte auf die Instanzvariablen und Methoden der inneren Klasse.
- Statisch geschachtelte Klassen werden häufig als Hilfsklassen für die Implementierung der äußeren Klasse eingesetzt.
- Beispiel: Implementierung einer Klasse als linear verkettete Liste ([siehe Kapitel 2](#)).

# Innere Klassen

---

```
class EnclosingClass {  
  
    class InnerClass{  
        ...  
    }  
  
    ... new InnerClass();  
}
```

- **Kopplung auf Instanzebene.**
- Wird in einem EnclosingClass-Objekt ein InnerClass-Objekt erzeugt, dann ist dieses InnerClass-Objekt mit dem EnclosingClass-Objekt gekoppelt und kann auf dessen Daten zugreifen.
- Wichtige Anwendungen:
  - Iteratoren (Iterator); siehe [Kapitel 5](#).
  - Beobachter (Listener) bei graphischen Benutzeroberflächen; siehe [Kapitel 12](#).

# Lokale innere Klassen

```
interface X { ... }

class A {

    X fun() {
        class Y implements X {
            ...
        }
        return new Y();
    }

    ...
}
```

Hier steht eine  
Implementierung des  
Interface X.

- Eine lokale innere Klasse ist eine Klasse, die innerhalb eines Methodenrumpfs definiert ist.
- Im Beispiel ist Y eine Klasse, die lokal in der Methode fun definiert ist. Das zurückgegebene Y-Objekt ist mit dem umfassenden A-Objekt gekoppelt.
- Anwendungen wie bei inneren Klassen. Beispiele in [Kapitel 12](#).

# Anonyme innere Klassen

- Speziell in grafischen Benutzeroberflächen werden oft mehrere und vor allem kleine innere Klassen (als Beobachter) benötigt.
- Statt jeweils eine innere Klasse mit Namen einzuführen, können auch anonyme Klassen (d.h. ohne Namen) definiert werden.
- Anwendungen wie bei inneren Klassen. Beispiele in [Kapitel 12](#).

```
interface X { ... }  
  
class A {  
    X fun() {  
        return new X() {  
            ...  
        }  
    }  
    ...  
}
```

Hier steht eine  
Implementierung des  
Interface X.

# Generische Typen

- Klassen und Interfaces lassen sich mit **Typparameter** versehen.

```
interface Liste<E> {  
    void add(E e);  
    E get(int index);  
}
```

E ist ein Typparameter.

```
class ListeAlsFeld<E> implements Liste<E>{ ... }
```

- Generische Typen lassen sich dann mit einer konkreten Klasse **instantiieren** (wie bei Aufruf von Methoden).

```
Liste<String> liste  
    = new ListeAlsFeld<String>();
```

Für E wird String als konkrete Klasse instantiiert.

- Wichtige Anwendung:  
Java Collections bestehen aus zahlreichen generischen Containern.
- Ausführliche Einführung in Kapitel 4.

# Funktionale Interfaces und Lambda-Ausdrücke

---

- **Lambda-Ausdrücke** (Funktionsausdrücke; seit Java 8).  
Damit können Funktionen als Parameter, als Rückgabewerte und als Zuweisungswerte verwendet werden (functions as first-class citizens).
- **Funktionales Interface** (= Interface mit genau einer abstrakten Methode) zur Typisierung von Lambda-Ausdrücken.
- **Beispiel:**

```
@FunctionalInterface  
interface Comparator<T> {  
    int compare(T x, T y);  
}
```

Lambda-Ausdruck

```
Comparator<Person> comp = (p1, p2) -> p1.getName().compareTo(p2.getName());  
persList.sort(comp);
```

- Details in Kapitel 13.