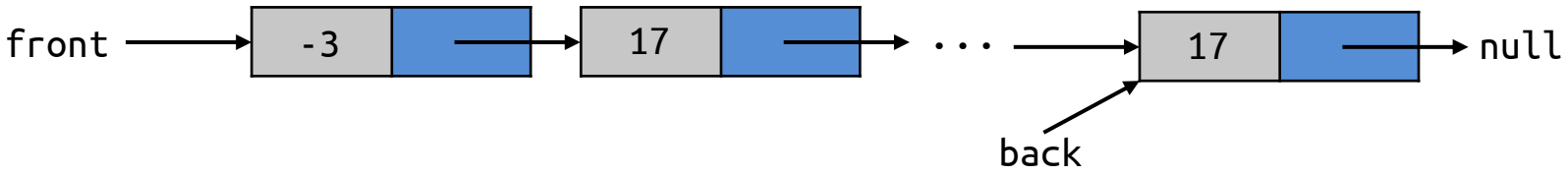
# 252-0027
# Einführung in die Programmierung

# 10. Vererbung

*Manuela Fischer, Malte Schwerhoff*

**Departement Informatik**
**ETH Zürich**

# Motivation: `FastLinkedList`




**Problem:** Einfügen am Ende einer `LinkedList` ist teuer: ganze Liste durchlaufen!

**Idee:** neue Klasse `FastLinkedList` mit einem Zeiger back auf das letzte Element.

**Frage:** Müssen wir nun alles nochmals implementieren/copy&pasten?

Code Reuse durch **Vererbung**

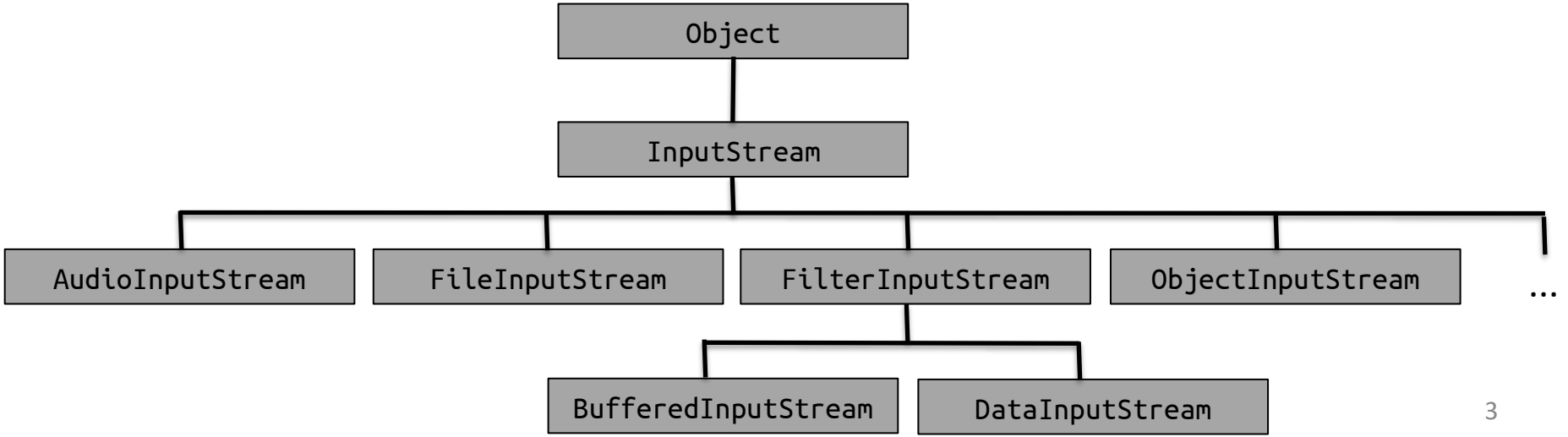
**Antwort:** Nein! Mit Vererbung kann Code wiederverwendet und erweitert werden.

# Motivation: Input-Stream in Java



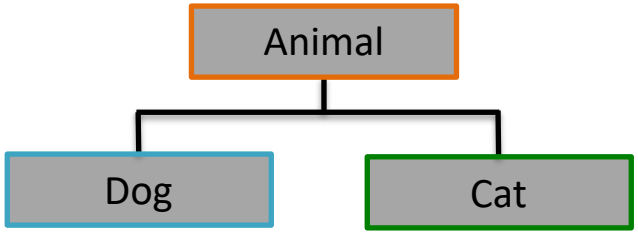
**Polymorphismus:**
Code funktioniert für unterschiedlichste Input-Quellen

`readData(InputStream source)`



# 10. Vererbung

# Klassisches Toy Example: Hund und Katze




- Gemeinsamkeiten
  - Eigenschaften: Name, Alter
  - Aktivitäten: essen, schlafen
- Hund-spezifisch
  - bellen, Briefträger (Postboten) beissen
- Katzen-spezifisch
  - miauen, Mäuse jagen

# 2 Optionen

A. **Kombiniert**
Je eine Klasse mit einem gemeinsamen und speziellen Teil

B. **Aufgeteilt**
eine allgemeine Klasse Tier und je eine Klasse mit Spezialisierung

Variante B. hat klare Vorteile:

- Anpassungen: Änderungen einfacher, da es nur 1 Stelle betrifft
- Lokalitätsprinzip: schnellerer Überblick über Spezialisierung
- Polymorphismus

# Klasse für Tier

```
public class Animal {
  public String name;
  public int age;
  public void eat() {
    System.out.println("Yummy!");
  }
  public void sleep() {
    System.out.println("Zzzzz!");
  }
}
```

Für den Moment kein Fokus auf Sichtbarkeit. Folgt später.

# Klassen für Hund und Katze (redundant!)

Code-Duplikation

```java
public class Dog {
  public String name;
  public int age;
  public void eat() {
    System.out.println("Yummy!");
  }
  public void sleep() {
    System.out.println("Zzzzz!");
  }
  public void bark() {
    System.out.println("Woof!");
  }
}
```

```java
public class Cat {
  public String name;
  public int age;
  public void eat() {
    System.out.println("Yummy!");
  }
  public void sleep() {
    System.out.println("Zzzzz!");
  }
  public void meow() {
    System.out.println("Meow!");
  }
}
```

# Vererbung

```java
public class Dog extends Animal {
  // everything that is in
  // class Animal and this:
  public void bark() {
    System.out.println("Woof!");
  }
}
```

```java
public class Cat extends Animal {
  // everything that is in
  // class Animal and this:
  public void meow() {
    System.out.println("Meow!");
  }
}
```

Nur spezifische Teile

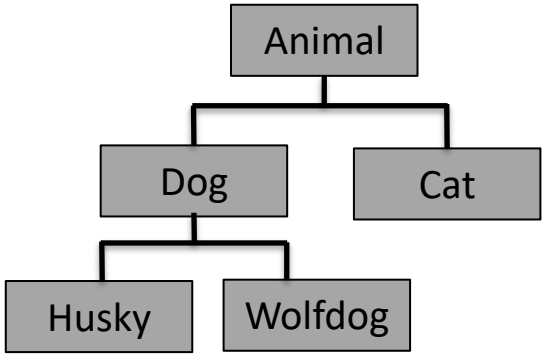
```
class B extends A
```

- Klasse B **erbt von** («**inherits from**») / **erweitert** («**extends**») Klasse A
  - Attribute/Methoden von A auch in B vorhanden: **Vererbung** («**inheritance**»)
  - Neue Attribute/Methoden können hinzugefügt werden: **Erweiterung** («**extension**»)
- B ist ein (Spezialfall von) A: **ist-ein-Beziehung** («**is-a relationship**»)
  - Wo Typ A gefragt ist, kann auch ein Objekt vom Typ B verwendet werden: **Subtyping**

Code Reuse!

folgt später!

# Vererbungshierarchie




**Vererbungshierarchie** («**inheritance hierarchy**»):
eine Menge von Klassen verbunden durch Vererbung

- Jede Klasse kann von höchstens einer anderen Klasse erben
  - Kein Mehrfachvererbung in Java!
- Viele Klassen können von der gleichen Klasse erben
- Beliebig lange **Vererbungsketten** («**inheritance chains**») möglich
- Terminologie:
  - Erbende Klassen: **Subklassen**/**Subtypen** («**subclasses**»/«**subtypes**»)
  - Vererbende Klassen: **Superklassen**/**Supertypen** («**superclasses**» / «**supertypes**»)

# 10. Vererbung

# Überschreiben von Methoden

- **Überschreiben** («**override**»): Definieren einer neuen Version einer Methode in einer Subklasse, die die Methode der Superklasse ersetzt
  - Bei Aufruf der Methode auf Objekt des Subtyps wird neue Methode ausgeführt (mehr Details dazu später!)
  - Muss gleiche Signatur und gleichen Rückgabetyp haben (sonst Fehler!)
  - Keine spezielle Syntax erforderlich: Methode neu definieren in Subklasse
  - (freiwillige) Annotation `@Override` drückt Absicht des Überschreibens aus

```
@Override
public returnType methodName(...) {
  ...
}
```

# Beispiel: Überschreiben von Methoden

```java
public class Animal {
  ...
  public void eat() {
    System.out.println("Yummy!");
  }
}
```

```java
public class Cat extends Animal {
  ...
  @Override
  public void eat() {
    System.out.println("Tuna, yay!");
  }
}
```

```java
Animal a = new Animal();
Cat c = new Cat();
Dog d = new Dog();
a.eat();
c.eat();
d.eat();
```

```
Yummy!
Tuna, yay!
Sausage!!
```

Jeweils Override ausgeführt!

```java
public class Dog extends Animal {
  ...
  @Override
  public void eat() {
    System.out.println("Sausage!!");
  }
}
```

# @Override ohne Override

```
public class Animal {
  public void eat() {
    System.out.println("Yummy!");
  }
}
```

```
public class Bird extends Animal {
  @Override // ERROR
  public void fly() {
    System.out.println("Flying!");
  }
  @Override // ERROR
  public void eat(String food) {
    System.out.println("eat " + food);
  }
}
```

Overload, nicht Override!

```
method does not override a
method from a supertype
```

Mit der freiwilligen `@Override`-Annotation gibt es einen Compilerfehler, falls keine entsprechende Methode (mit gleicher Signatur) in der Superklasse existiert.

# Override versus Overload

- **Overloading (Methodenüberladung):** (unabhängig von Vererbung)
  Mehrere Methoden mit dem gleichen Namen aber anderer Signatur
- **Override (Methodenüberschreibung):** (nur im Kontext von Vererbung)
  Methode mit gleicher Signatur* wird in Subtyp überschrieben

```
public class Animal {
  public void eat() {
    System.out.print("Yummy!");
  }
}
```

```
public class Cat extends Animal {
  public void eat(String food) {
    System.out.print("eat" + food);
  }
}
```

```
Animal().eat();        // Yummy!
Cat().eat("tuna");     // eat tuna
Cat().eat();           // Yummy!
```

Cat hat 2 Overloads der Methode eat, eine geerbt von Animal, eine neu definiert.

* Rückgabetyp und Sichtbarkeit müssen übereinstimmen

# Mit der Superklasse arbeiten

```
super.methodName(parameters)
```

- Subklasse kann Methoden der Superklasse aufrufen
  - Für jede Methode möglich, nicht nur für gerade überschriebene
  - Referenzvariable `super` bezieht sich auf *dieses Objekt* als Instanz der *Superklasse*
  - Zum Vergleich und als Erinnerung:
    Referenzvariable `this` bezieht sich auf *dieses Objekt* als Instanz der *Subklasse*
  - Für nicht überschriebene Methoden sind die folgenden Aufrufe äquivalent:

```
super.methodName(parameters);
this.methodName(parameters);
methodName(parameters);
```

# Beispiele: Mit Superklasse arbeiten

```java
public class Animal {
  public void eat() {
    System.out.print("Yummy!");
  }
}
```

```java
Animal a = new Animal();
Cat c = new Cat();
Dog d = new Dog();
a.eat();
c.eat();
d.eat();
```

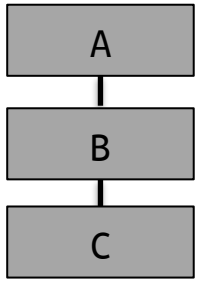
```
// Yummy!
// Tuna, yay! Yummy!
// Yummy! Sausage!
```

```java
public class Cat extends Animal {
  @Override
  public void eat() {
    System.out.print("Tuna, yay! ");
    super.eat(); // Animal.eat()
  }
}
```

```java
public class Dog extends Animal {
  @Override
  public void eat() {
    super.eat(); // Animal.eat()
    System.out.print(" Sausage!");
  }
}
```

# Superklassen und Vererbungsketten

```
public class A {}
public class B extends A {}
public class C extends B {}
```




- C erbt Attribute/Methoden von B, die wiederum von A geerbt sind
- In B überschriebene oder ergänzte Methoden werden so von C übernommen
- C kann wiederum die geerbten Methoden überschreiben
- `super` bezieht sich jeweils auf direkten Vorgänger

# Beispiel: Superklassen mit Vererbungsketten

```java
public class Animal {
  public void eat() {
    System.out.print("Yummy!");
  }
}
```

```java
public class Dog extends Animal {
  @Override
  public void eat() {
    super.eat(); // Animal.eat()
    System.out.print(" Sausage!");
  }
}
```

```java
Animal a = new Animal();
Dog d = new Dog();
Husky h = new Husky();
a.eat();
d.eat();
h.eat();
```

```java
public class Husky extends Dog {
  @Override
  public void eat() {
    super.eat(); // Dog.eat()
    System.out.println(" Steak!");
  }
}
```

```
// Yummy!
// Yummy! Sausage!
// Yummy! Sausage! Steak!
```

# 10. Vererbung

# Zur Erinnerung: Konstruktoren

- Für jede Klasse gibt es automatisch den parameterlosen Default Constructor

```
class A {
  int x;
  String y;
}
```

```
A() {
  x = 0;
  y = null;
}
```

- Falls ein Konstruktor definiert wird, so gibt es keinen Default Constructor
  - Parameterloser Konstruktor müsste dann explizit definiert werden!

# Vererbung und Default Constructors

```java
class A {
  int x;
  String y;
}
```

```java
public A() { // default constructor
  x = 0;
  y = null;
}
```

```java
class B extends A {
  int z;
}
```

```java
public B() { // default constructor
  // implicit call of A()
  z = 0;
}
```

Default Constructor der Subklasse B ruft automatisch zuerst den Default Constructor der Superklasse A auf.

# Beispiel: Vererbung und Default Constructors

```java
public class Animal {
  public String name;
  public int age;
  public void sleep() {...}
  public void eat() {...}
  // default constructor Animal()
}
```

```java
// implicit default constructor
public Animal() {
  name = null;
  age = 0;
}
```

```java
public class Cat extends Animal {
  int mice; // # caught mice
  // default constructor Cat()
}
```

```java
// implicit default constructor
public Cat() {
  // implicit call of Animal()
  mice = 0;
}
```

# Vererbung und Konstruktoren

- Konstruktoren werden *nicht* vererbt!
  - Subklassen erhalten nur den parameterlosen Default Constructor
- Konstruktor der Subklasse ruft Konstruktor der Superklasse auf, entweder
  a. Expliziter Aufruf als *erste* Anweisung im Konstruktor der Subklasse mit `super(...);`
  b. Oder automatischer Aufruf des Default Constructors vor Konstruktor-Rumpf

```
class B extends A {
  public B(int x) {
    super(x); // call of A(x)
    ...
  }
}
```

Option a.

```
class B extends A {
  public B(int x) {
    // implicit call of A()
    ...
  }
}
```

Option b.

# Beispiel Vererbung und Konstruktoren

```java
public class Animal {
  public String name;
  public int age;
  public void sleep() {...}
  public void eat() {...}

  public Animal(String n) {
    name = n;
    age = 0;
  }
}
```

```java
public class Cat extends Animal {
  int mice; // # caught mice

  public Cat(String n) {
    super(n);
    mice = 0;
  }
}
```

# Vorsicht mit Vererbung und Konstruktoren

```java
public class Animal {
  public String name;
  public int age;
  public void sleep() {...}
  public void eat() {...}

  public Animal(String n) {
    name = n;
    age = 0;
  }
}
```

```java
public class Cat extends Animal {
  int mice; // # caught mice
  // default constructor Cat()
}
```

```java
// implicit default constructor
public Cat() {
  // implicit call of Animal() ERROR!
  mice = 0;
}
```

```
Cat.java:2: cannot find symbol
symbol  : constructor Animal()
location: class Animal
public class Cat extends Animal {
       ^
```

Animal hat keinen Default Constructor (mehr)!

# Vorsicht mit Vererbung und Konstruktoren

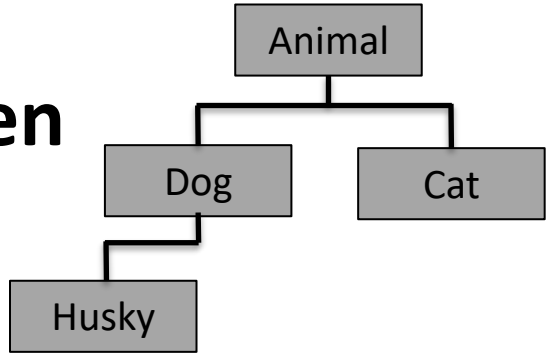
**Technisch:**
Falls Superklasse einen Konstruktor (mit Parametern) definiert, muss

a. sie explizit einen parameterlosen Konstruktor definieren

b. oder ihre Subklassen auch einen Konstruktor definieren

**Intuitiv:**
Falls Superklasse etwas Spezielles für die Superklasse-Instanzen festlegen will (durch einen Konstruktor mit Parametern), so sollte das auch für die Subklasse-Instanzen so passieren.

# Vererbungsketten und Konstruktoren




```java
public class Animal {
  public String name;
  public Animal(String n) {
    name = n;
  }
}
```

```java
public class Dog extends Animal {
  public Dog() {
    super("Bello");
  }
}
```

```java
public class Husky extends Dog {
  // implicit default constructor
}
```

```java
Animal("Bello");
```

expliziter Aufruf

```java
Dog();
```

impliziter Aufruf

```java
Husky h = new Husky();
```

`super(...)` ruft Konstruktor des direkten Vorgängers in der Hierarchie auf!

# 10. Vererbung

# Sichtbarkeit: Überblick

Informationen zu Packages folgen! Annahme im Moment (und meistens): Alles im gleichen Package!

- Sichtbarkeit von Klassen
  a. `public class A { ... }`
     - Überall sichtbar
     - `import packageName.A` falls in anderem Package
  b. `class A { ... }`
     - Default
     - Nur im Package sichtbar
     - Muss nicht zwingend in Datei `A.java` sein
  c. `private` innere Klassen
     - Nur in der äusseren Klasse sichtbar

Es gibt noch einige nicht-empfohlene (und für uns irrelevante) Möglichkeiten.

- Sichtbarkeit von Attribut/Methode
  a. `public T m() { ... }`
     - Überall sichtbar
  b. `protected T m() { ... }`
     - Im Package und in allen Subklassen (auch in anderen Packages) sichtbar
  c. `T m() { ... }`
     - Default
     - Nur im Package sichtbar
  d. `private T m() { ... }`
     - Nur in der Klasse sichtbar

# Sichtbarkeit: `private`

```
public class Animal {
  private String name; // prevent client access
}
```

```
public class Cat extends Animal {
  public void eat() {
    System.out.println(name + " eats tuna!"); // ERROR
  }
}
```

The field Animal.name is not visible.

`private` Attribute sind in Subklasse nicht (direkt) zugreifbar

- Stichwort Encapsulation!

# `private` und Vererbung?

```
public class Animal {
  private String name;
  public Animal(String name) {
    this.name = name;
  }
  public void eat() {
    System.out.print(name + " eats");
  }
}
```

```
public class Cat extends Animal {
  public Cat(String name) {
    super(name);
  }
}
```

```
Cat c = new Cat("Chebyshev");
c.eat(); // OK
```

```
Chebyshev eats
```

Geerbte Methoden können auf `private` Attribute zugreifen!

# private und Overriding?

```
public class Animal {
  private String name;
  public void eat() {
    System.out.print(name + " eats");
  }
}
```

```
public class Cat extends Animal {
  @Override
  public void eat() {
    System.out.print(name + " eats tuna"); // ERROR
  }
}
```

The field Animal.name is not visible.

Beim Überschreiben von Methoden darf nicht auf `private` Attribute zugegriffen werden!

# `private` und Overriding!

```java
public class Animal {
  private String name;
  public void eat() {
    System.out.print(name + " eats");
  }
}
```

```java
public class Cat extends Animal {
  @Override
  public void eat() {
    super.eat(); // call Animal.eat()
    System.out.print(" tuna"); // OK
  }
}
```

Alternative:
Getter `getName()`
verwenden!

# `private` und (kein) Overriding

Subklasse kann Methode aus der Superklasse nur überschreiben, falls Methode in Subklasse sichtbar ist

- Falls `private` (oder `default` in anderem Package): nicht sichtbar!
- Falls nicht sichtbar: kein Override, sondern neue Methode!

```
public class A {
  public  void f() { g(); }
  private void g() { ...  }
}
```

```
public class B extends A {
  private void g() { ...  }
  public  void h() { g(); }
}
```

Kein Override

```
A a = new A();
B b = new B();
a.f();  // call A.f & A.g
b.h();  // call B.h & B.g
b.f();  // call A.f & A.g
```

A.f ruft immer A.g und nie B.g auf (später mehr)!

# Sichtbarkeit: `protected`

```
public class Animal {
  protected String name;
}
```

```
public class Cat extends Animal {
  public void eat() {
    System.out.println(name + " eats tuna!"); // OK
  }
}
```

`protected` Attribute sind in Klasse und Subklassen direkt zugreifbar!

- Sowohl lesend als auch schreibend!
- Sinnvoll, wenn nur Subklassen Zugriff brauchen, andere Klassen nicht!

# Sichtbarkeit nicht einschränken bei Overrides!

`public > protected > default > private`

Bei Override darf Sichtbarkeit nur vergrössert werden (oder gleich bleiben), nicht verkleinert werden!

- Instanz der Subklasse ist Instanz der Superklasse (ist-ein-Beziehung): Methoden aus Superklasse müssen für Subklasse verfügbar bleiben.

Cannot reduce the visibility of the inherited method from A!

```
public class A {
  public void a() { ... }
  protected void b() { ... }
  void c() { ... }
  private void d() { ... }
}
```

```
public class B extends A {
  void a() { ... }          // ERROR
  public void b() { ... }   // OK
  private void c() { ... }  // ERROR
  public void d() { ... }   // OK
}
```

# 10. Vererbung

# Wiederholung: Klassen, Typen und Referenzen

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

- Klassen definieren einen (neuen) Typ

```
A a;
```

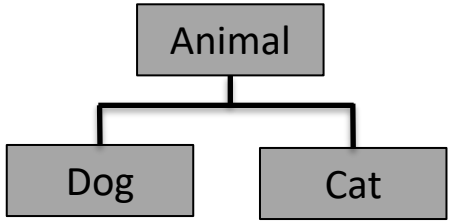
- Variablen sind Referenzen auf Instanzen dieses Typs

```
A a1 = new A();
A a2 = a1; // alias
```

# Subtyping

```
public class Animal { ... }
public class Cat extends Animal { ... }
public class Dog extends Animal { ... }
```

```
Animal a1 = new Animal();
Animal a2 = new Cat();
a2 = new Dog();
```

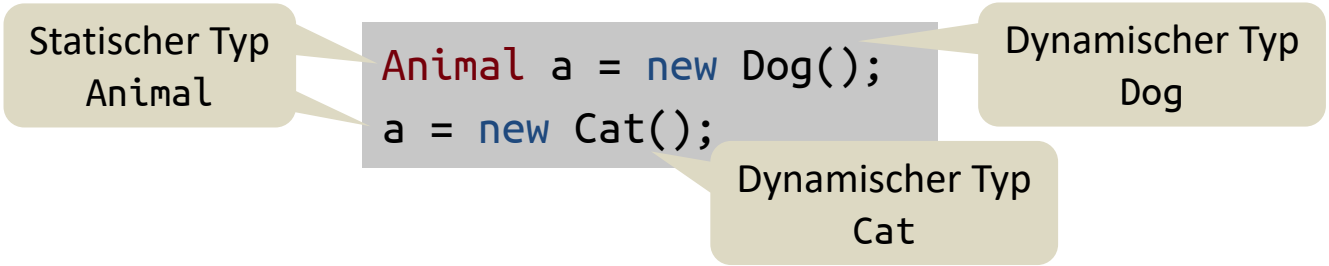



**Statischer** Typ `Animal`,
**dynamischer** Typ `Cat`

- Referenzvariablen der Superklasse können auf Objekte der Subklasse zeigen!
- **Subtyping**: Überall wo Objekt der Superklasse gefragt ist (Zuweisung, Methodenparameter), kann auch Objekt der Subklasse verwendet werden (ist-ein-Beziehung).

Impliziter Upcast

# Statischer und Dynamischer Typ




- **Statischer Typ** («**static type**») einer Variable (was der Compiler sieht):
  bei der Deklaration einer Variable angegebener Typ (im Code)
- **Dynamischer Typ** («**dynamic type**») einer Variable:
  Typ des Objekts, das die Variable (momentan) referenziert (zur Laufzeit)
- Konsequenzen von Subtyping:
  - Dynamischer Typ ist Subtyp des statischen Typs (oder der gleiche Typ).
  - Statischer Typ bleibt; dynamischer Typ kann sich ändern (durch Umhängen der Referenz auf neues Objekt).

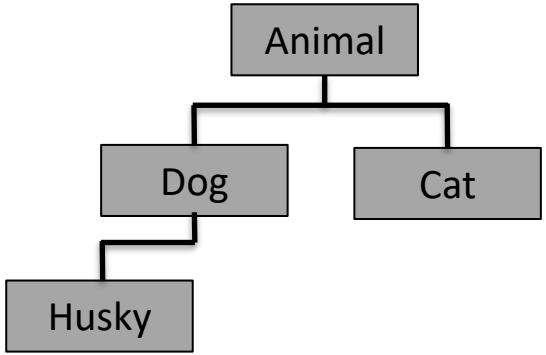
# Subtyping: Umkehrung?

```
public class Animal { ... }
public class Cat extends Animal { ... }
public class Dog extends Animal { ... }
```

```
Dog d = new Animal();   // COMPILER ERROR
Cat c = new Dog();      // COMPILER ERROR
Husky h = new Dog();    // COMPILER ERROR
```

```
Animal a = new Husky();
Dog d = a;              // COMPILER ERROR
```

*Incompatible types*




Compiler kann nicht (immer) wissen, dass `Animal` auf einen Husky zeigt

- Referenzvariablen der Subklasse können **nicht** auf Objekte der Superklasse zeigen: ist-ein-Beziehung ist asymmetrisch!
- Statischer Typ der rechten Seite muss Subtyp des linken Typs sein

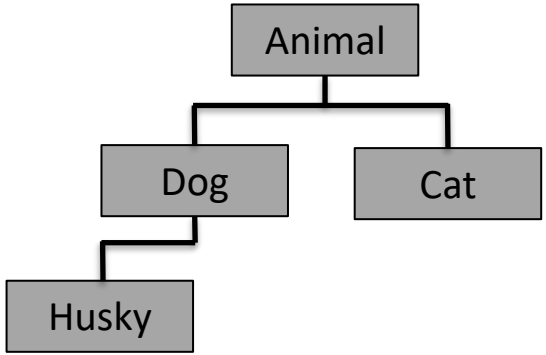
# Type Cast (Expliziter Downcast)

```
(B) a
```

- Für Referenzvariable `a` vom statischen Typ `A` und `B` Subklasse von `A`
- Programm versucht zur Laufzeit die Referenz `a` vom Typ `A` in eine Referenz vom Typ `B` umzuwandeln
  a. `a` zeigt auf ein Objekt vom Typ `B` (oder Subtyp): erfolgreicher Type Cast
     - Compiler darf jetzt annehmen, dass `a` auf ein Objekt vom Typ `B` zeigt
     - `(B) a` ist eine Referenzvariable vom statischen Typ `B`
  b. `a` zeigt nicht auf ein Objekt vom Typ `B` (oder Subtyp): Laufzeitfehler
- Objekt, auf das `a` zeigt, bleibt unverändert
  - Unterschied zu Casts für primitive Typen, die Werte ändern (können)

# Type Casts: Beispiele

```
Animal a = new Animal();
Animal c = new Cat();
Animal d = new Dog();
Animal h = new Husky();
Dog dd = new Dog();
```

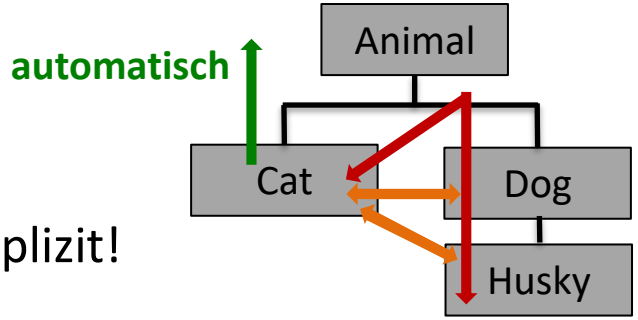



```
Animal a1 = (Animal) c;          // OK, but pointless
Cat c1 = (Cat) c;                // OK!
Dog d1 = (Dog) d;                // OK!
Dog d2 = (Dog) h;                // OK!
Husky h1 = (Husky) h;            // OK!
Husky h2 = (Husky) d;            // RUNTIME ERROR
Cat c2 = (Cat) d;                // RUNTIME ERROR
Cat c3 = (Dog) d;                // COMPILER ERROR
Cat c4 = (Cat) dd;               // COMPILER ERROR
```

Type Cast erfolgreich, aber Dog kann nicht Cat zugewiesen werden

Dog kann nie eine Cat sein, Type Cast nicht erlaubt!

# Type Casts: Typische Fehler




- Nach oben casten ist unnötig; Upcast geschieht implizit!

```
Cat c = new Cat();
Animal a = (Animal) c; // OK, but not needed!
```

- Nach unten casten ergibt Laufzeitfehler, falls weiter nach unten als dyn. Typ

```
Animal a = new Dog();
Husky h = (Husky) a; // RUNTIME CAST ERROR
```

- Nach unten casten ergibt Laufzeitfehler, falls in anderen Branch als dyn. Typ

```
Animal a = new Dog();
Cat c = (Cat) a; // RUNTIME CAST ERROR
```

- Seitlich casten ergibt Compilerfehler (*inconvertible types*)

```
Cat c = new Cat();
Dog d = (Dog) c; // COMPILER ERROR
```

# Type Checks mit `instanceof`-Operator

```
variable instanceof type
```

- Ausdruck mit Typ `boolean` und Wert `true`
  falls `variable` auf Objekt von Typ `type` (oder Subtyp) zeigt.
  - `null instanceof type` ist `false` für alle Typen

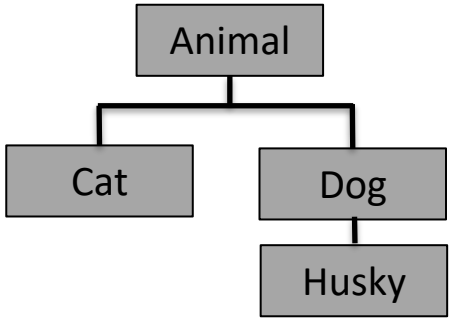
```
String s = "Hi";
Animal a = new Animal();
Cat c = new Cat();
```

| instanceof | String | Animal | Cat |
|---|---|---|---|
| s | true | false | false |
| a | false | true | false |
| c | false | true | true |
| null | false | false | false |

# Type Checks: Beispiele

```
Animal a = ...;
```

```
if (a instanceof Dog) {
  Dog d = (Dog) a;
  if (d instanceof Husky) {
    Husky h = (Husky) d;
  }
} else if (a instanceof Cat) {
  Cat c = (Cat) a;
} else {
  System.out.println("Neither a dog nor a cat!");
}
```



# Zusammenfassung: Subtyping

Instanz der Subklasse ist eine Instanz der Superklasse (ist-ein-Beziehung)

- Referenzvariable der Superklasse kann auf Instanz der Subklasse zeigen


- Nicht umgekehrt: Compilerfehler

```
Dog d = new Animal(); // COMPILER ERROR
```

- Umgekehrt nur mit Type Casts (Laufzeitfehler falls nicht möglich!)

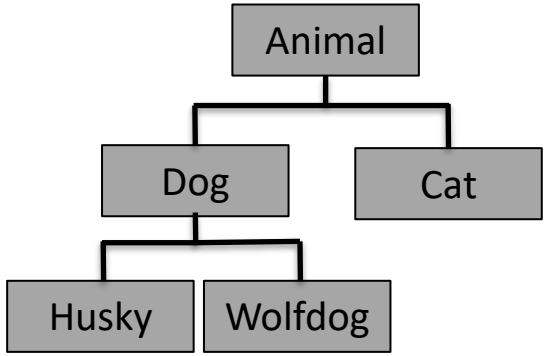
```
Animal a = new Dog();
Dog d = (Dog) a; // OK, Type Cast
Cat c = (Cat) a; // RUNTIME ERROR
Cat e = a instanceof Cat ? (Cat) a : null; // OK, e == null
```

# Subtyping mit Methodenparametern!

```
public class DogShelter {
  ...
  public void adopt(Dog d) { ... }
}
```

Formaler Parameter:
Statischer Typ Dog

```
DogShelter shelter = ...;
Husky h = ...;
shelter.adopt(h); // OK!
Wolfdog w = ...;
shelter.adopt(w); // OK!
```



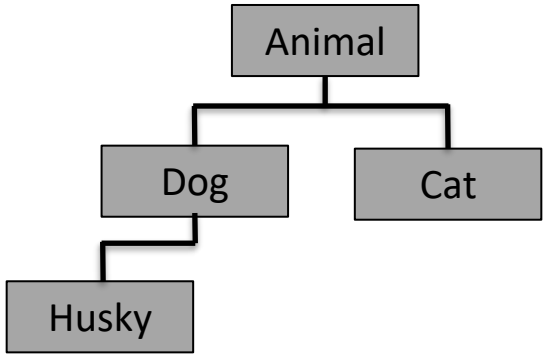

```
Animal a = new Husky();
shelter.adopt(a); // COMPILER ERROR
```

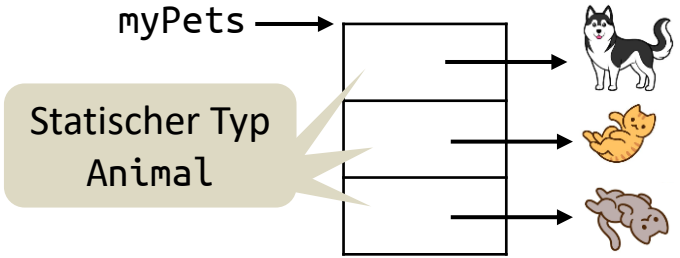
- Formaler Parameter eines Supertyps kann (Referenzen auf) Objekte beliebiger Suptypen als Argumente haben.
- Statischer Typ des übergebenen Arguments muss Subtyp des Typs des formalen Parameters sein (gleiche Regeln wie bei Zuweisung).

# Subtyping mit Arrays!

```
public class Animal { ... }
public class Cat extends Animal { ... }
public class Dog extends Animal { ... }
public class Husky extends Dog { ... }
```

```
Animal[] myPets = new Animal[3];
myPets[0] = new Husky("Awena");
myPets[1] = new Cat("Bruschetta");
myPets[2] = new Cat("Chebyshev");
```




- Array des Supertyps kann (Referenzen auf) Objekte beliebiger Subtypen enthalten!

# Beispiel und Frage

```java
public class Animal {
  public void eat() {
    System.out.print("Yummy!");
  }
}
```

```java
Animal[] myPets = new Animal[2];
myPets[0] = new Dog();
myPets[1] = new Cat();
for (int i = 0; i < 2; i++) {
  myPets[i].eat();
}
```

Was ist der Output?

```java
public class Cat extends Animal {
  @Override
  public void eat() {
    System.out.print("Tuna, yay! ");
    super.eat(); // Animal.eat()
  }
}
```

```java
public class Dog extends Animal {
  @Override
  public void eat() {
    super.eat(); // Animal.eat()
    System.out.println(" Sausage!");
  }
}
```

# Methoden und statischer Typ

- Der statische Typ einer Variablen entscheidet darüber, welche Methoden auf dem referenzierten Objekt aufgerufen werden können!
- Der dynamische Typ ist irrelevant: Methoden, die nur in der Klasse des Objekts definiert sind, sind nicht aufrufbar!
  - Referenzvariable des Supertyps erlaubt keinen Zugriff auf im Subtyp definierte Methoden und Attribute!

Statischer Typ Dog

Statischer Typ Animal

```
Dog d = new Dog();
d.bark(); // outputs Woof!
Animal a = d;
a.bark(); // ERROR
```

Animal hat keine Methode bark

# Methoden und Type Casts

- Mit Referenzvariablen können wir nur Methoden, die im statischen Typ der Variable definiert sind, aufrufen (Typ des Objekts irrelevant).

```
Animal a = new Cat();
a.meow(); // COMPILER ERROR
```

  - Compiler-Logik: Referenz a könnte auf beliebiges Tier zeigen, z.B. auch auf Hund

- Type Cast nötig für Objekt-spezifische Methoden!

```
Animal a = new Cat();
Cat c = (Cat) a;
c.meow();
```

Explizite Referenzvariable c

```
Animal a = new Cat();
// same thing in one line:
((Cat) a).meow();
```

Anonyme Referenzvariable

# Dynamische Bindung bei Overrides

Statischer Typ
Animal

```
Animal a = new Dog();
a.eat(); // Dog-override
```

Dynamischer Typ
Dog

**Dynamische Bindung** («**dynamic binding**» / «**dynamic dispatch**»):
Bei Aufruf einer Methode mit Overrides entscheidet der Typ des Objekts, auf der die Methode aufgerufen wird, (also der dynamische Typ der Variable) über die Override-Version!

Details folgen!

Achtung: Dynamische Bindung gilt nur für Objekt, auf dem die Methode aufgerufen wird, nicht für Parametervariablen (dort ist nur der statische Typ entscheidend)!

# Zurück zum Beispiel

```
public class Animal {
  public void eat() {
    System.out.print("Yummy!");
  }
}
```

```
Animal[] myPets = new Animal[2];
myPets[0] = new Dog();
myPets[1] = new Cat();
for (int i = 0; i < 2; i++) {
  myPets[i].eat();
}
```

```
Yummy! Sausage!
Tuna, yay! Yummy!
```

```
public class Cat extends Animal {
  @Override
  public void eat() {
    System.out.print("Tuna, yay! ");
    super.eat(); // Animal.eat()
  }
}
```

```
public class Dog extends Animal {
  @Override
  public void eat() {
    super.eat(); // Animal.eat()
    System.out.println(" Sausage!");
  }
}
```

# Übersicht: Methodenresolution

```
Dog d = new Dog();
Animal a = d;
```

Kurzversion; Details auf kommenden Folien!

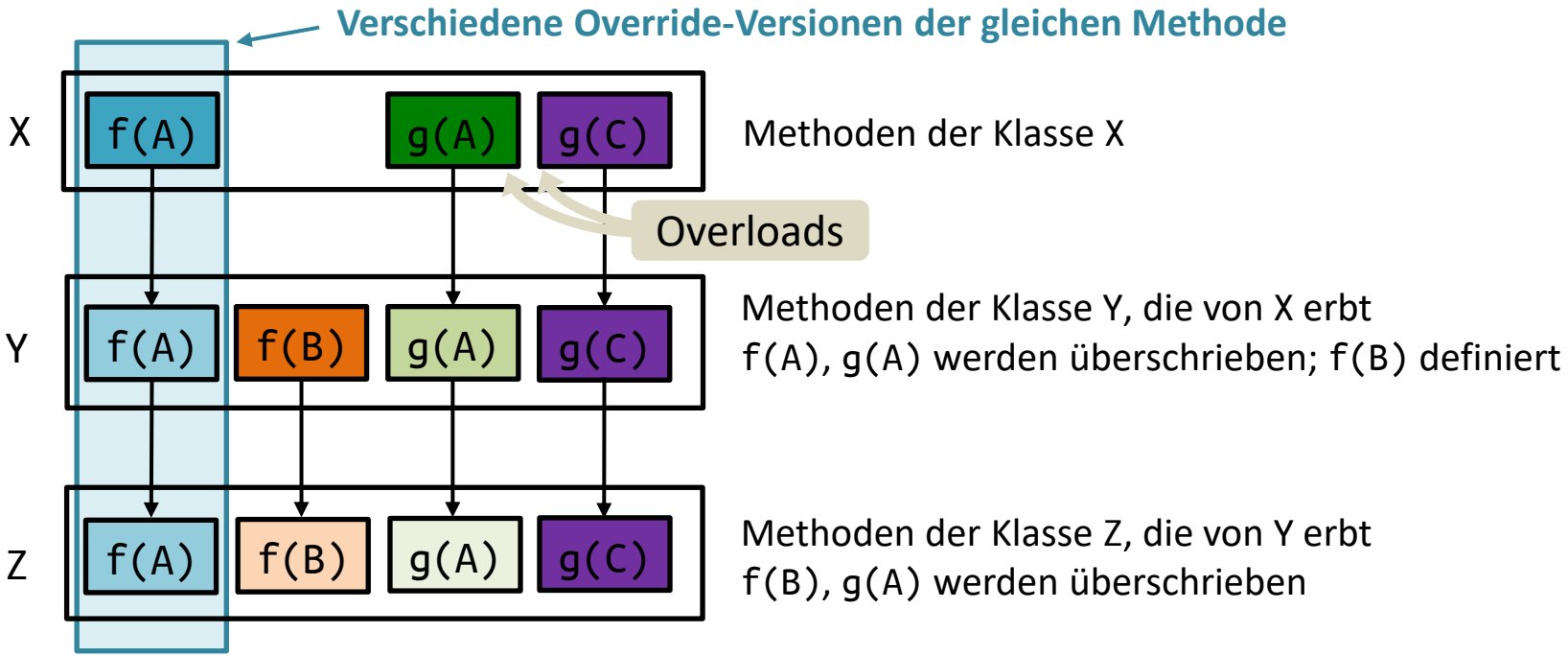
1. **Zur Compile-Zeit:** Statischer Typ entscheidet, ob passende Signatur existiert

```
d.eat();  // OK: class Dog has (inherited) method eat
a.eat();  // OK: class Animal has defined method eat
d.bark(); // OK: class Dog has defined method bark
a.bark(); // ERROR: class Animal does not have method bark
```

2. **Zur Laufzeit:** Dynamischer Typ entscheidet, welche Version ausgewählt wird, falls mehrere Overrides der Methode mit dieser Signatur existieren

```
d.eat();  // Dog-override of eat is called
a.eat();  // Dog-override of eat is called
d.bark(); // method bark defined in class Dog is called
```

# Schematische Übersicht: Methodenresolution

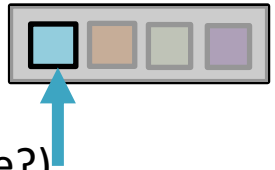



1. Horizontal: Compiler entscheidet mit statischem Typ, ob Methode mit Signatur existiert
2. Vertikal: Zur Laufzeit wird Override-Version aufgrund vom dynamischen Typ ausgewählt

# Details: 1. Auswahl der Signatur (Overload)

Compiler kennt nur statische (deklarierte) Typen

1. **Zur Compile-Zeit: Auswahl der Signatur (welcher Overload?)**
   - Statischer Typ des Aufruferobjekts und Sichtbarkeit entscheiden, welche Methoden existieren/zugreifbar sind
   - Auswahl der Methode aufgrund der statischen Typen der Argumente (gibt es eine passende Methode; welcher Overload ist der passendste?)



     a. Passen die Typen direkt?
     b. Passen die Typen nach
        - impliziter Konvertierung? (für primitive Typen, z.B. `int` ⟶ `double`)
        - bzw. implizitem Upcast? (für Referenztypen, z.B. `Cat` ⟶ `Animal`)
     c. Passen die Typen nach (Un)boxing (+ eventuell b.)? (z.B. `int` ⟷ `Integer`)

stark vereinfacht; mehr Details hier!

   - Gibt es keine passend(st)e Methode, so gibt es einen Compilerfehler

# (Un)Boxing: Böse Überraschungen

a. Passen die Typen direkt?

b. Passen die Typen nach
   - impliziter Konvertierung? (für primitive Typen, z.B. `int` ⟶ `double`)
   - bzw. implizitem Upcast? (für Referenztypen, z.B. `Cat` ⟶ `Animal`)

c. Passen die Typen nach (Un)boxing (+ eventuell b.)? (z.B. `int` ⟷ `Integer`)

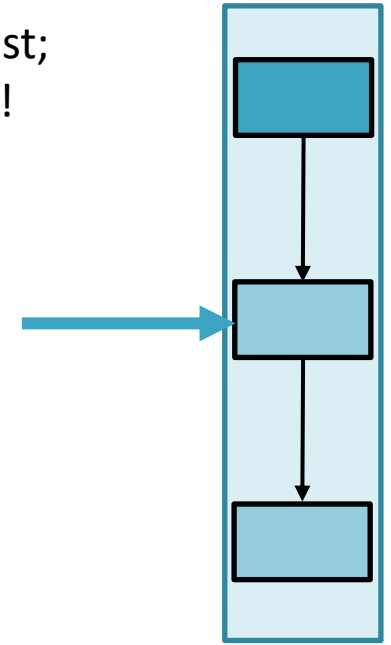
- `Integer` ⟶ `int` ⟶ `double` erlaubt!
- `Integer` ⟶ `int` ⟶ `double` ⟶ `Double` nicht erlaubt!
- `int` ⟶ `double` ⟶ `Double` nicht erlaubt!
- `int` ⟶ `Integer` ⟶ `Object` erlaubt (`Object` ist Supertyp von jedem Referenztyp; später mehr)
- `Integer` ⟶ `Object` Vorrang vor `Integer` ⟶ `int`

**Wrapper-Typen und Autoboxing selten (und bewusst) einsetzen!**

# Details: 2. Bindung der Methode (Override)

**2. Zur Laufzeit: Dynamische Bindung der Methode (welcher Override?)**

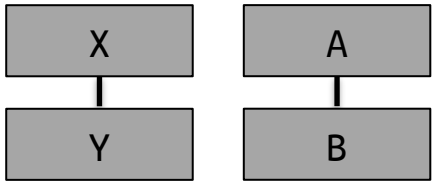
- Nach Kompilieren steht Signatur der Methode (Overload) fest; nur über Implementation (Override) wird noch entschieden!
- Dynamischer Typ des Aufruferobjekts entscheidet, welche Version der Methode ausgewählt wird, falls mehrere Versionen (Overrides) existieren.
- Dynamische Typen der Argumente spielen keine Rolle!



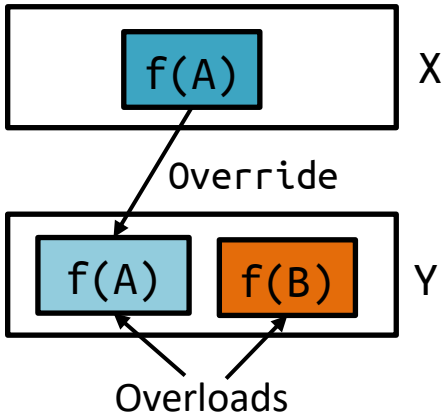
# Schematisches Beispiel 1

```
A a  = new A();
B b  = new B();
A ab = new B();
```

```
X x  = new X();
Y y  = new Y();
X xy = new Y();
```




```
x.f(a);   // f(A) in X
x.f(b);   // f(A) in X (B is an A)
x.f(ab);  // f(A) in X
```



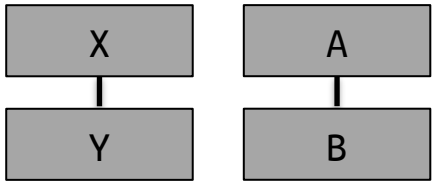

```
y.f(a);   // f(A) in Y
y.f(b);   // f(B) in Y
y.f(ab);  // f(A) in Y (static type of arg.)
```

```
xy.f(a);   // f(A) in Y
xy.f(b);   // f(A) in Y (B is an A)
xy.f(ab);  // f(A) in Y
```

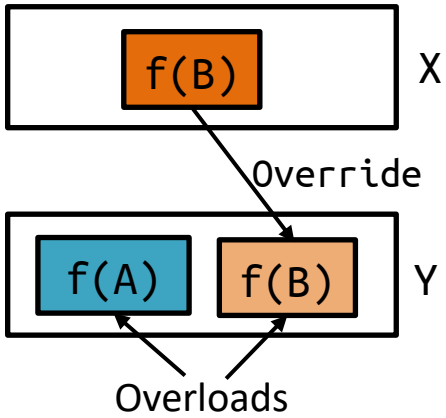
# Schematisches Beispiel 2

```
A a  = new A();
B b  = new B();
A ab = new B();
```

```
X x  = new X();
Y y  = new Y();
X xy = new Y();
```




```
x.f(a);   // ERROR, no method X.f(A)
x.f(b);   // f(B) in X
x.f(ab);  // ERROR, no method X.f(A)
```




```
y.f(a);   // f(A) in Y
y.f(b);   // f(B) in Y
y.f(ab);  // f(A) in Y (static type of arg.)
```

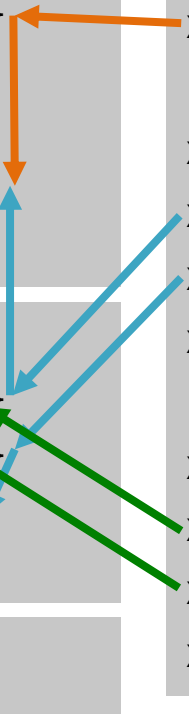
```
xy.f(a);   // ERROR, no method X.f(A)
xy.f(b);   // f(B) in Y
xy.f(ab); // ERROR, no method X.f(A)
```

# Beispiel 1: Variante 1

```
class T {
  void s0() { m(); }
  void s1() { ... }
  void s2() { ... }
  void m()  { ... }
}
```

```
class S extends T {
  void s1() { m(); }
  void s2() { p(); }
  void p()  { ... }
}
```

```
class R extends S {
  void p()  { ... }
}
```



```
T x = new T();  // static & dynamic type T
x.s0();

x = new S();    // static T & dynamic S
x.s1();
x.s2();
x.p();          // COMPILER ERROR

x = new R();    // static T & dynamic R
x.s1();
x.s2();
x.p();          // COMPILER ERROR
```

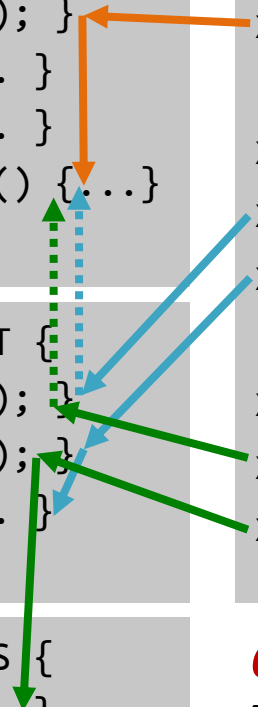
# Beispiel 1: Variante 2

```
class T {
  void s0() { m(); }
  void s1() { ... }
  void s2() { ... }
  private void m() {...}
}
```

```
class S extends T {
  void s1() { m(); }
  void s2() { p(); }
  void p()  { ... }
}
```

```
class R extends S {
  void p()  { ... }
}
```

```
T x = new T();  // static & dynamic type T
x.s0();

x = new S();    // static T & dynamic S
x.s1();
x.s2();

x = new R();    // static T & dynamic R
x.s1();
x.s2();
```



***COMPILER ERROR***

Methode `m()` ist nicht definiert für S (und R) wenn `m()` in T Sichtbarkeit `private` hat.

# Beispiel 1: Variante 3

```
class T {
  void s0() { m(); }
  void s1() { ... }
  void s2() { ... }
  void m() {...}
}
```

```
class S extends T {
  void s1() { m(); }
  void s2() { p(); }
  void p()  { ... }
}
```

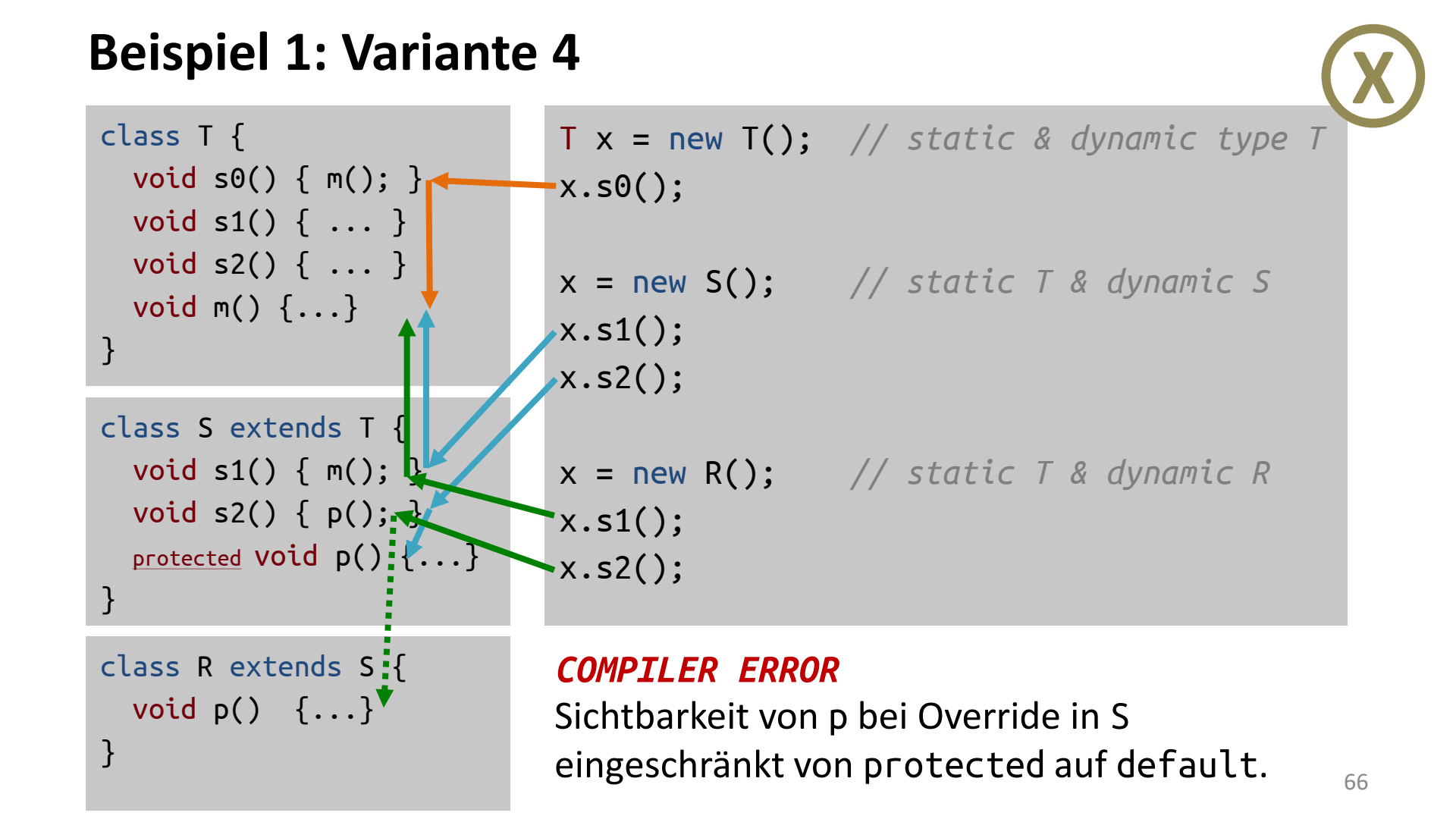
```
class R extends S {
 private void p() {...}
}
```

```
T x = new T();  // static & dynamic type T
x.s0();

x = new S();    // static T & dynamic S
x.s1();
x.s2();

x = new R();    // static T & dynamic R
x.s1();
x.s2();
```

***COMPILER ERROR***

Sichtbarkeit von p bei Override in S eingeschränkt von `default` auf `private`.

# Beispiel 1: Variante 4



```
class T {
  void s0() { m(); }
  void s1() { ... }
  void s2() { ... }
  void m() {...}
}
```

```
class S extends T {
  void s1() { m(); }
  void s2() { p(); }
  protected void p() {...}
}
```

```
class R extends S {
  void p()  {...}
}
```

```
T x = new T();  // static & dynamic type T
x.s0();

x = new S();    // static T & dynamic S
x.s1();
x.s2();

x = new R();    // static T & dynamic R
x.s1();
x.s2();
```

***COMPILER ERROR***

Sichtbarkeit von p bei Override in S eingeschränkt von `protected` auf `default`.

# Beispiel 2: Teil 1

```
public class Snow {
  public void method2() {
    System.out.print("Snow2 ");
  }
  public void method3() {
    System.out.print("Snow3 ");
  }
}
```

```
public class Sleet extends Snow {
  public void method2() {
    System.out.print("Sleet2 ");
    super.method2();
    method3();
  }
  public void method3() {
    System.out.print("Sleet3 ");
  }
}
```

```
public class Rain extends Snow {
  public void method1() {
    System.out.print("Rain1 ");
  }
  public void method2() {
    System.out.print("Rain2 ");
  }
}
```

```
public class Fog extends Sleet {
  public void method1() {
    System.out.print("Fog1 ");
  }
 public void method3() {
    System.out.print("Fog3 ");
  }
}
```

# Beispiel 2: Teil 2

```
Snow var1 = new Sleet();
var1.method2();
```

Sleet2 Snow2 Sleet3

```
Snow var2 = new Fog();
var2.method2();
```

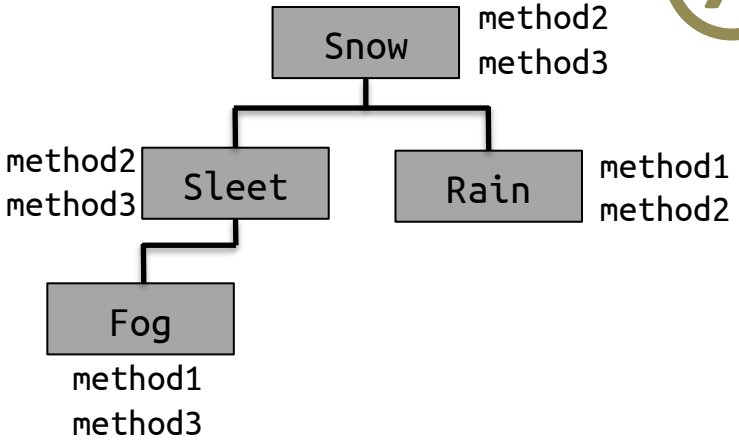
Sleet2 Snow2 Fog3

```
Snow var3 = new Rain();
var3.method1();
```

COMPILER ERROR

```
Snow var3 = new Rain();
((Fog) var3).method1();
```

RUNTIME ERROR




| | Snow | Rain | Sleet | Fog |
|---|---|---|---|---|
| method1 | - | Rain1 | - | Fog1 |
| method2 | Snow2 | Rain2 | Sleet2<br>Snow2<br>Sleet3 | Sleet2<br>Snow2<br>Fog3 |
| method3 | Snow3 | Snow3 | Sleet3 | Fog3 |

# Overloads und Unboxing

```
void foo(int x)       { /* A */ }
void foo(double x)    { /* B */ }
void foo(Integer x)   { /* C */ }
void foo(Double x)    { /* D */ }
void foo(Object x)    { /* E */ }
```

```
int      a = 1;
double   b = 2;
Integer  c = a;
Object   d = c;
```

```
foo(a);
foo(b);
foo(c);
foo(d);
```

Hier greift jeweils Fall 1 (Typen passen direkt):

- `foo(a)` → A
- `foo(b)` → B
- `foo(c)` → C
- `foo(d)` → E

# Overloads und (Un)boxing

```
void foo(double x)    { /* B */ }
void foo(Integer x)   { /* C */ }
```

```
int      a = 1;
double   b = 2;
Integer  c = a;
Object   d = c;
```

```
foo(a);
foo(b);
foo(c);
foo(d);
```

- `foo(a)` → B (Fall 2: Implizite Konvertierung)
- `foo(b)` → B (Fall 1: Direkt passend)
- `foo(c)` → C (Fall 3: Direkt passend)
- `foo(d)` → Compilerfehler

# Overloads und Unboxing

```
void foo(Integer x)   { /* C */ }
void foo(Object x)    { /* E */ }
```

```
int      a = 1;
double   b = 2;
Integer  c = a;
Object   d = c;
```

```
foo(a);
foo(b);
foo(c);
foo(d);
```

- `foo(a)` → C (Fall 3: Boxing)
- `foo(b)` → E (Fall 3, dann Fall 2 (Upcast): `double` ⟶ `Double` ⟶ `Object`)
- `foo(c)` → C (Fall 1: Direkt passend)
- `foo(d)` → E (Fall 1: Direkt passend)

# Polymorphismus

**Polymorphismus** («**polymorphism**»): Programm ist so entwickelt, dass es für unterschiedliche Objektypen verwendet werden kann und sein Verhalten an diese Typen anpasst.

- Etymologie: **poly** (viele) + **morphe** (Gestalt, Form)

Arten von Polymorphismus

- Methodenüberladung (Overloading)
- Objektorientierter Polymorphismus (OOP): Subtyping und Dynamic Binding
- Generics (später!)

# OOP-Polymorphismus und Parameter

Wenn Methode als Parameter eine Referenzvariable der Klasse T erwartet, so kann auch Referenzvariable einer Subklasse von T übergeben werden.

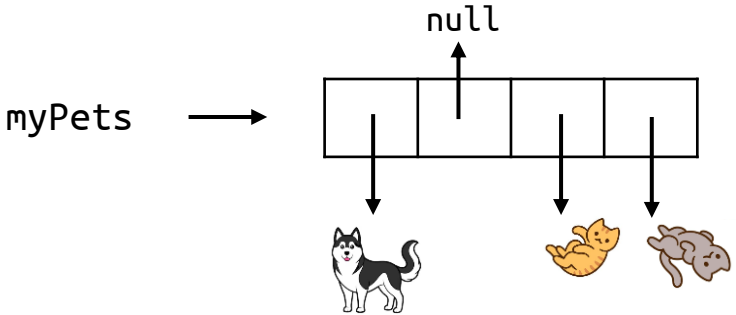
- Methode lässt Referenzen auf T und alle Subtypen zu (**Subtyping**)!

```
public static void printInfo(Animal a) {
   ...
}
```

- Durch **Dynamic Binding** wird Verhalten auf den dynamischen Typ angepasst!

# OOP-Polymorphismus und Arrays

Ein Array der Klasse T kann auch Referenzvariablen auf Subklassen von T speichern!




- T[ ] lässt Referenzen auf T und alle Subtypen zu (**Subtyping**)!
- Durch **Dynamic Binding** wird Verhalten auf den dynamischen Typ angepasst!

# Randbemerkung: Statische Methoden und Dynamic Binding

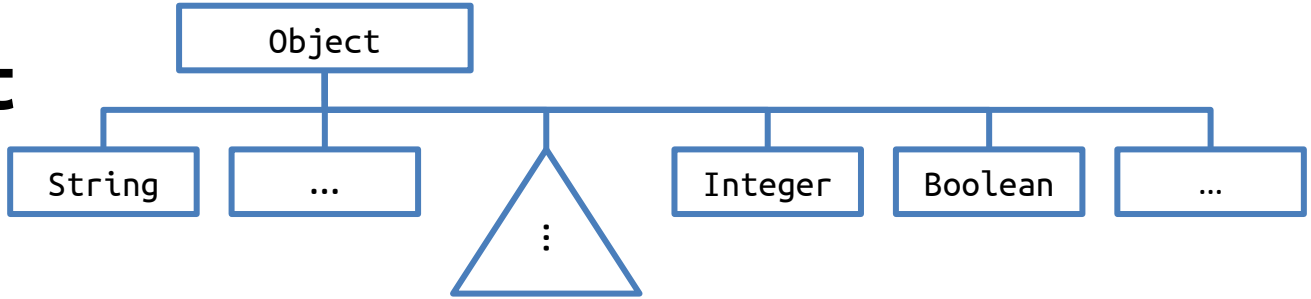
- Statische Methoden können nur von statischen Methoden überschrieben werden
  - Können, sollten aber nicht!

- Statische Methoden brauchen keine Instanz
  - Kein dynamischer Typ
  - Statischer Typ entscheidet

```
A.f(); // original
B.f(); // override
```

# 10. Vererbung

# Klasse `Object`




- Besondere Klasse `java.lang.Object`
  - `Object` hat keine Superklasse
  - Zuoberst in der Vererbungshierarchie, über allen Klassen
  - Jede Klasse ist implizit Subklasse (direkt oder indirekt) von `Object`
  - Referenzvariable vom Typ `Object` kann auf alle Objekte zeigen

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

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

Dürfte man so schreiben!

# Methoden der Klasse `Object`

| Methoden von `Object` |
|---|
| `equals` |
| `toString` |
| `getClass` |
| `hashCode` |
| `notify` |
| `wait` |

- `public String toString()`
  - Liefert Textdarstellung des Objekts (für Ausgabe)
- `public boolean equals(Object other)`
  - Vergleicht impliziter Parameter `this` mit `other`

- Jede Klasse erbt diese Methoden!
  - Name, Rückgabetyp, Parametertypen und Sichtbarkeit sind vorgegeben
  - Methoden machen nicht immer, was wir wollen

Später mehr!

# Beispiele: `Object`-Variablen

```
public class Animal { ... }
```

```
Object o1 = new Animal();
Object o2 = "Hello there!";
Object o3 = new Scanner(System.in);
```

```
String s = o1.toString();              // OK
int l = o2.length();                   // ERROR
double d = o3.nextDouble();            // ERROR
```

Nicht definiert für Klasse `Object`

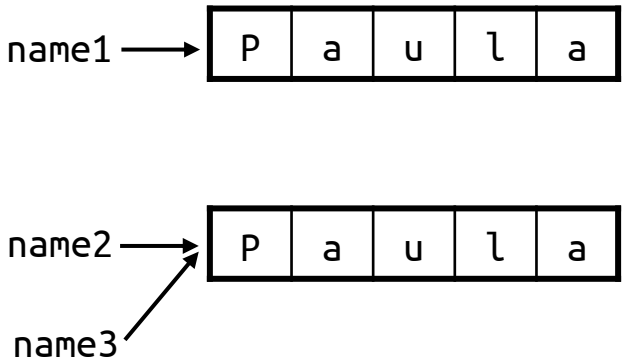
---

```
public static boolean isNull(Object o) {
  return o == null;
}
```

Methode akzeptiert alle Objekte!

# Vergleich von Objekten: So nicht!

```
String name1 = "Paula";
String name2 = "Paula";
if (name1 == name2) // false
  System.out.println("Same name!");
String name3 = name2;
if (name2 == name3) // true
  System.out.println("Same name!");
```

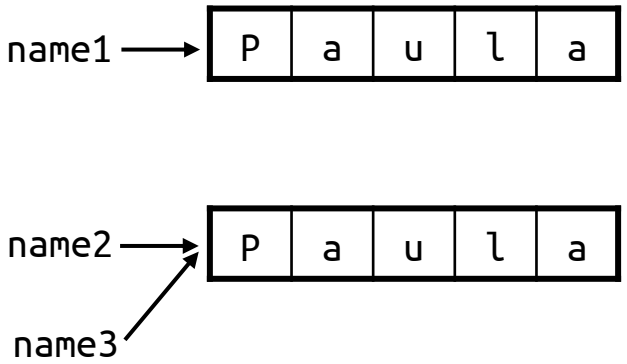



- Der Operator == funktioniert nicht wie gewünscht mit Objekten, sondern nur mit primitiven Typen.
  - == vergleicht Referenzen (Speicheradressen), nicht Inhalt/Zustand der Objekte
  - == liefert nur dann `true`, wenn die zwei Operanden auf das gleiche Objekt zeigen

Manchmal wollen wir Referenzen (Speicheradressen) vergleichen. Überlegen Sie sich ein Beispiel!

# Vergleich von Objekten: So geht es!

```
String name1 = "Paula";
String name2 = "Paula";
if (name1.equals(name2)) // true
  System.out.println("Same name!");
String name3 = name2;
if (name2.equals(name3)) // true
  System.out.println("Same name!");
```

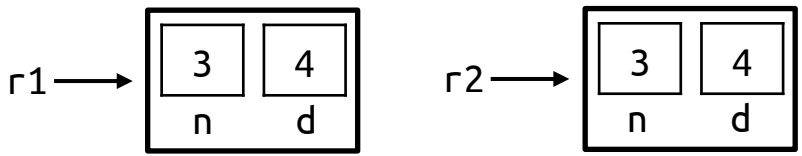



- Die Methode `equals` vergleicht Inhalt/Zustand der Objekte
  - Definiert in Klasse `Object`
  - Wie weiss `Object`-Methode, wie man Strings vergleicht?!?

# Vergleich von beliebigen Objekten mit `equals`?

```
public class Rational {
  int n; // numerator
  int d; // denominator
  public Rational(int n, int d) {...}
}
```

```
Rational r1 = new Rational(3, 4);
Rational r2 = new Rational(3, 4);
if (r1.equals(r2)) { // false
  System.out.println("same number");
}
```




- `equals`-Methode kann nicht den Zustand von Objekten vergleichen
  - kann nicht wissen, wie beliebige Objekte verglichen werden sollen
  - Attributweiser Vergleich oft nicht gewünscht:
    `Rational(3,4).equals(Rational(6,8))`
- Geerbte Methode `equals` verhält sich (vorerst) wie `==`
  - Kann überschrieben werden (`@Override`)

# Vergleich von beliebigen Objekten mit equals?

```
public class Rational {
  ...
  public boolean equals(Rational other) {
    return n * other.d == d * other.n;
  }
}
```

Vergleicht nun Zustand der Rational-Objekte

```
Rational r1 = new Rational(3, 4);
Rational r2 = new Rational(6, 8);
if (r1.equals(r2)) { // true
  System.out.println("same number");
}
```

Funktioniert wie erwartet!

```
same number
```

```
Object o = new Rational(3, 4);
Rational r = new Rational(6, 8);
if (o.equals(r)) { // false
  System.out.println("same number");
}
```

Funktioniert nicht! Erklärung folgt!

# Vergleich von beliebigen Objekten mit `equals`?

```
public class Rational {
  ...
  @Override // ERROR
  public boolean equals(Rational other) {
    return n * other.d == d * other.n;
  }
}
```

- `equals(Rational other)` ist kein Override von `equals(Object other)`
  - Parametertypen dürfen nicht verändert werden bei Overrides!
  - Neue Methode, die nur für statischen Typ `Rational` (oder Subtyp) aufrufbar ist!

# Vergleich von beliebigen Objekten mit `equals`?

```
public class Rational {
  ...
  @Override // OK
  public boolean equals(Object other) {
    return n * other.d == d * other.n; // ERROR
  }
}
```

Notwendig für Override!

Klasse `Object` hat keine Attribute `n` und `d`.

```
Rational(3, 4).equals("Hello!");
Rational(3, 4).equals(Animal());
```

Kann mit beliebigen Parametertypen aufgerufen werden!

Override von `equals(Object other)` in Klasse `Rational` soll immer `false` zurückgeben, falls other nicht dynamischen Typ `Rational` hat!

# equals mit Type Casts?

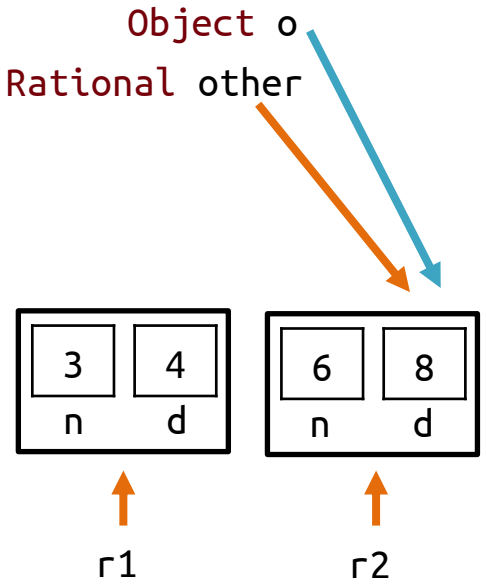
```
public class Rational {
  ...
  @Override
  public boolean equals(Object o) {
    Rational other = (Rational) o; // Type Cast
    return n * other.d == d * other.n;
  }
}
```

```
Rational r1 = new Rational(3, 4);
Rational r2 = new Rational(6, 8);
if (r1.equals(r2)) { // true
  System.out.println("same number");
}
```

```
same number
```

Referenzvariable vom (statischen) Typ

- **Object**
- **Rational**



# equals mit Type Casts?

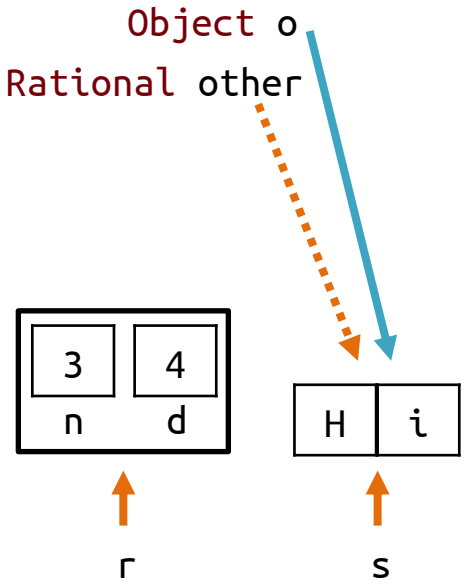
```
public class Rational {
  ...
  @Override
  public boolean equals(Object o) {
    Rational other = (Rational) o; // Type Cast
    return n * other.d == d * other.n;
  }
}
```

**String cannot be cast to class Rational**

```
Rational r = new Rational(3, 4);
String s = "Hi";
if (r.equals(s)) { // RUNTIME ERROR
  System.out.println("same number");
}
```

Referenzvariable vom (statischen) Typ

- **Object**
- **Rational**



# `instanceof`-Operator und `Object`

```
Object s = "Hi";
Object c = new Cat();
```

| instanceof | String | Cat | Object |
|---|---|---|---|
| s | true | false | true |
| c | false | true | true |
| null | false | false | false |

Für Referenzvariable `a` gilt `(a instanceof Object) == (a != null)`

# Lösung für equals-Methode

```
public class Rational {
  ...
  @Override
  public boolean equals(Object o) {
    if (o isinstanceof Rational) {          // Type Check
      Rational other = (Rational) o;        // Type Cast
      return n * other.d == d * other.n;    // comparison of Rationals
    } else {                                // not a Rational: not equal!
      return false;
    }
  }
}
```

# toString-Methode

- Definiert in Klasse `Object`
  - Gibt Adresse und Typ zurück
  - Eigene Version durch Überschreiben (Override)

```
public class Rational {
  ...
  @Override
  public String toString() {
    return n + " / " + d;
  }
}
```

`@Override` beschützt uns vor Tippfehlern wie z.B. `tostring()`

# 10. Vererbung

# Was passiert hier?

```java
class A {
  int x;
}
```

```java
class B extends A {
  boolean x;
}
```

```java
A a = new A();
a.x = 6;          // OK
a.x = true;       // COMPILER ERROR: TYPE MISMATCH
```

```java
A a = new B();
a.x = 6;          // OK
a.x = true;       // COMPILER ERROR: TYPE MISMATCH
```

```java
B b = new B();
b.x = 6;          // COMPILER ERROR: TYPE MISMATCH
b.x = true;       // OK
```

# Verstecken von Attributen


```
class A {
  int x;
  int a;
}
```

```
class B extends A {
  boolean x;
  double b;
}
```


**Verstecken** («**shadowing**»): Das Attribut einer Subklasse versteckt das Attribut der Superklasse mit dem gleichen Namen.

- Subklasse erbt Attribut, aber es ist versteckt durch anderes Attribut
- Kein Überschreiben (Overriding) von Attributen!

# Zugriff auf versteckte Attribute


```java
class A {
  int x;
  int a;
}
```

```java
class B extends A {
  boolean x;
  double b;
}
```

```java
B b = new B();
System.out.println(b.x);            // false
System.out.println(((A) b).x);      // 0
A a = b;
System.out.println(a.x);            // 0
```

Auf verstecktes Attribut kann nur via Superklasse zugegriffen werden!

a. Upcast: **`(A) b`**

b. Subtyping: Zuweisen zu einer Variable vom statischen Typ **A**

Impliziter Upcast

# Attribute versus Methoden

**Methoden**

- Können überschrieben werden (Override)
- Dynamischer Typ entscheidet über Version (Dynamic Binding)

**Attribute**

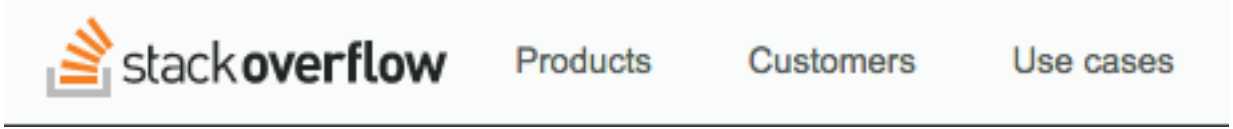
- Können versteckt werden (Shadowing)
  - Kann man, sollte man aber nicht!
- Statischer Typ entscheidet über Version

# Attribute versus Methoden: Beispiel

```
class A {
  String s = "in A";
  String myS() { return s; }
}
```

```
class B extends A {
  String s = "in B";
  @Override
  String myS() { return s; }
}
```

```
B b = new B();
System.out.println(b.s);            // in B
System.out.println(b.myS());        // in B
System.out.println(((A) b).s);      // in A (static type!)
System.out.println(((A) b).myS());  // in B (dynamic type!)
```




- One major purpose [of these shadowing rules] is to confuse people. It's bad practice and should be avoided.
- If you can avoid it [shadowing] though, you should since it may cause confusion.

# 10. Vererbung

10.1 Vererbung, Vererbungshierarchie, Vererbungsketten

10.2 Overriding von Methoden

10.3 Konstruktoren und Vererbung

10.4 Sichtbarkeit

10.5 Subtyping, Dynamic Binding und Polymorphismus

10.6 Klasse `Object`; Methoden `equals` und `toString`

10.7 Shadowing von Attributen

**10.8 Überblick und Ausblick**

# Vererbung: Übersicht

- Attribute
  - erben
  - hinzufügen
  - verstecken (kann man, sollte man aber vermeiden!)
- Methoden
  - erben
  - hinzufügen
  - abändern (Override)
  - auf ursprünglicher Version aufbauen (aufrufen) mit `super`
  - aber nicht löschen oder Sichtbarkeit einschränken (sonst Compilerfehler) oder Signatur ändern (sonst Overload statt Override)!
- Konstruktoren
  - Werden nicht vererbt!
  - Konstruktor der Superklasse aufrufen mit `super()`

# Was wenn ich nicht möchte, dass…

- … meine Methode überschrieben wird? → finale Methode
- … von meiner Klasse geerbt wird? → finale Klasse
- … meine Klasse instanziiert wird? → abstrakte Klasse
  (oder private Konstruktoren)

# Das Keyword `final`: Teil 1

`final` Variable/Attribut darf nicht auf linker Seite der Zuweisung stehen!

- Für primitive Typen: unveränderbar!

```
final double pi = 3.14159;
pi = 3.14; // COMPILER ERROR: cannot assign a value to final variable
```

- Für Referenztypen: Referenz (aber nicht Objekt!) unveränderbar!
  - Referenz zeigt für immer auf das gleiche Objekt (kein Umhängen des Zeigers)

```
final Cat c = new Cat();
c.name = "Minka"; // OK, object can be changed
c = new Cat(); // COMPILER ERROR
```

# Das Keyword `final`: Teil 2

- Für Klassen: kein Erweitern möglich!

```
final class A { ... }
class B extends A { ... } // COMPILER ERROR: cannot extend final class
```

- Für Methoden: kein Override möglich!

```
class A {
  final void m() { ... }
}
class B extends A {
  void m() { ... } // COMPILER ERROR: cannot override final method
}
```

# Abstrakte Klassen

- Abstrakte Klassen können nicht instanziiert werden
  - Keyword `abstract`
  - Referenzvariablen möglich, aber sie zeigen auf Instanzen von Subtypen
- Abstrakte Klassen können abstrakte Methoden ohne Rumpf haben
  - Implementation bei nicht-abstrakten Subklassen

```
abstract class Animal {
  public abstract void talk();
  public void eat() { ... }
}
```

```
class Cat extends Animal {
  @Override
  public void talk() { ... }
}
```

```
Animal a = new Animal();   // COMPILER ERROR!
Animal a = new Cat();      // OK!
```

# Beispiel

```
abstract class Shape {
  Position p; // class Position not shown here
  double ratio() {
    return perimeter() / area();
  }
  abstract double area();
  abstract double perimeter();
}
```

Gemeinsamkeiten werden implementiert (vorgegeben).

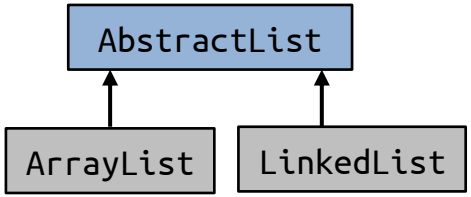
Spezifische Sachen werden offen gelassen (Lücken).

```
class Circle extends Shape {
  double radius;
  double area() { return ... }
  double perimeter() { return ... }
}
```

```
class Square extends Shape {
  double length;
  double area() { return ... }
  double perimeter() { return ... }
}
```

# Wann abstrakte Klassen?

- Keine Instanzen dieser Klasse, aber Instanzen von Subklassen erwünscht!
- Partielle Lösungen mit Lücken vorhanden; Lücken sollen von Subklassen gefüllt werden




- Beispiel:
  Abstrakte Klasse `AbstractList`;
  Konkrete Klassen `ArrayList` und `LinkedList`
  `AbstractList` implementiert Gemeinsamkeiten und lässt Lücken offen;
  Spezifisches wird in konkreten Subklassen implementiert!



Thema wird später im Kontext von Interfaces nochmals aufgegriffen!

# Wer erbt von wem?

a. 2-dimensionale und 3-dimensionale Punkte?

```
class Point2D {
  int x;
  int y;
}
```

```
class Point3D extends Point2D {
  int z;
}
```

b. 2-dimensionale Punkte und Kreise?

```
class Point2D {
  int x;
  int y;
}
```

```
class Circle extends Point2D {
  int r;
}
```

# Wer erbt von wem? (Fortsetzung)

c. Quadrate und Rechtecke?

```
class Square {
  int a;
}
```

```
class Rectangle extends Square {
  int b;
}
```

d. Rechtecke und Quadrate?

```
class Rectangle {
  int a;
  int b;
}
```

```
class Square extends Rectangle {
  // INV: a == b
}
```

# Wer erbt von wem? (Fortsetzung 2)

e. Mengen und unveränderbare Mengen?

```
class MutableSet {
  ArrayList<Integer> elements;
  MutableSet(...) {...}
  int size() { ... }
  boolean cotains(int e) { ... }
  void add(int e) { ... }
  void remove(int e) { ... }
}
```

```
class ImmutableSet {
  ArrayList<Integer> elements;
  ImmutableSet(...) {...}
  int size() { ... }
  boolean cotains(int e) { ... }
}
```

# Wann soll man (nicht) Vererbung verwenden?

- **Liskovsches Substitutionsprinzip** («**Liskov substitution principle**»):
  Objekt einer Subklasse kann überall *erfolgreich* verwendet werden,
  wo ein Objekt der Superklasse erwartet wird.
  - Is-a relationship: Subklassen-Objekt ist Spezialfall von Superklasse
    - 3D-Punkt ist kein 2D-Punkt
    - Kreis ist kein 2D-Punkt
    - Rechteck ist kein Quadrat
    - Quadrat ist zwar ein Rechteck, aber …
    - Unveränderbare Menge ist zwar eine Menge, aber …
  - Alle Attribute/Methoden der Superklasse existieren auch in Subklasse
    - … zweite Seitenlänge des Rechtecks hätte keine Bedeutung für Quadrat
    - … Methoden `add`/`remove` wäre nicht erlaubt für `ImmutableSet`
  - Code-Wiederverwendung alleine noch kein Argument für Vererbung!
    - Code-Redundanz ist besser als unintuitive Vererbungshierarchien

Barbara Liskov ist eine US-amerikanische Informatikerin. Sie gewann 2008 den Turing-Award.

# Komposition statt Vererbung

- Oft handelt es sich um eine **hat-ein-Beziehung** («**has-a relationship**»)
  - Kreis hat einen (Mittel-)Punkt.

- Komposition/Aggregation («composition»/«aggregation») statt Vererbung
  - Referenzattribut auf andere Klasse

```
class Circle ~~extends Point2D~~ {
  Point2D center;
}
```

Kreis **hat** einen (Mittel-)Punkt.
Keine Vererbung!