

```
class Article {
```

← Oberklasse / Basisklasse

```
    int code;  
    int price;
```

```
    boolean available() {...}
```

```
    void print() {...}
```

```
} Article(int code, int price) {...}
```

```
class Book extends Article { ← Unterklasse
```

```
    String author;
```

```
    String title;
```

@Override

```
void print() {...}
```

erbt: code, price, available, print

überschreibt: print

ergänzt: author, title

```
Book(int code, int price, String author, String title) {
```

.....

```
}
```

Article

code

price

available()

print()

Article(c,p)



Book

author

title

print()

Book()

Überschreiben von Methoden

```
class Article {
    ...
    void print() {
        Out.print(code + " " + price);
    }
}
```

```
Article(int c, int p) {
    code = c; price = p;
}
```

```
class Book extends Article {
```

```
...
void print() {
    super.print();
    Out.print(" " + author + " " + title);
}
```

```
Book(int c, int p, String a, String t) {
    super(c, p);
    author = a; title = t;
}
```

Benutzung:

Book book = new Book(code, price, author, title)

→ erzeugt Book-Objekt
→ Book-Konstruktor

→ Article-Konstruktor(code
| price = p, author = a,
| title = t)

book →

code
price
author
title

book.print();

→ print als Book

→ parent als Article code:price

→ Out.print author:title

@ Overwrite ⇒ A-Notation

Weerschrijven van Konstruktorer

Kein Konstruktor
in Oberklasse

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

```
class B {  
    B(int x) {...}  
}
```

```
B b = new B(3)
```

Parameterloser Konstruktor
in Oberklasse

```
class A {  
    A() {}  
}
```

```
class B extends A {  
    B(int x) {...}  
}
```

```
B b = new B(3);
```

Konstruktor mit
Parametern in Oberklasse

```
class A {  
    A(int y) {}  
}
```

```
class B extends A {  
    B(int x) {}  
}
```

```
B b = new B(3);
```

OK

Aufruf von:

- A()
- Standardkonstruktor
- B(int x)

OK

Aufruf von

- A()
- B(int x)

Fehler!

Es wurde kein
Standardkonstruktor
Aberzeugt. Man
muss schreiben

```
class B extends A {  
    B(int x) {super(x);}  
}
```

Zuweisungskompatibilität

Oberklassenvariable = Unterklassenvariable,

Jeder ClockTimer ist ein Timer → ClockTimer-Objekt können in Timer-Variablen gespeichert werden.

Was geschieht bei Zuweisung

clockTimer clockTimer

= new ClockTimer(...);

clockTimer

hrs
min
sec
maxHrs

Timer timer = clockTimer;

clockTimer

hrs
min
sec
maxHrs

timer

- Timer zeigt auf gleicher Objekt wie clockTimer
- nur Timer-Felder sind über timer zugänglich:

timer.hrs = ... 110k
timer.maxHrs = ... 11 Fehler
clockTimer.maxHrs = ... 11 ok

Illegaler Zuweisung

Unterklassenvariable = Oberklassenvariable; // Verloren

(Nicht jeder Timer ist ein ClockTimer)

Was wäre wenn sonst die Folge?

Timer timer = new Timer(...); clockTimer

hrs
min
sec

clockTimer clockTimer = timer;

timer

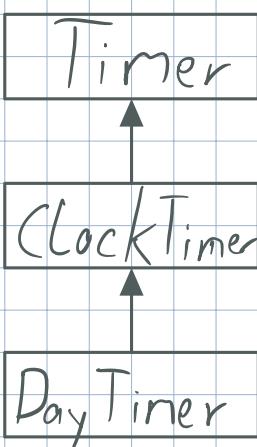
hrs
min
sec
? ?

clockTimer

- clockTimer würde auf gleiches Objekt wie immer zeigen

- clockTimer.maxHrs undefined

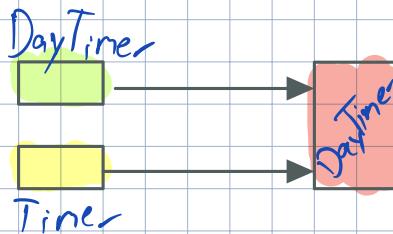
Statischen & dynamischer Typ



DayTimer dayTimer = new DayTimer(..);

Timer timer = dayTimer;

ClockTimer clockTimer = dayTimer;
timer = clockTimer;



Statischer Typ	Dynamische Typ
----------------	----------------

DayTimer DayTimer

Timer DayTimer

ClockTimer DayTimer

Timer DayTimer

Typ der Variable Typ des Objekt

Statischer Typ => Allgemeiner oder gleich dem Typ des Obj.

Laufzeit-Typkasten

Dynamischer Typ kann abgefragt werden

if (timer instanceof ClockTimer) {

}

if (timer != null && timer.getClass()

== ClockTimer.class)

Timer timer = new Timer(...);
ClockTimer clockTimer = new ClockTimer(...);
DayTimer dayTimer = new DayTimer(...);

if (!clockTimer instanceof ClockTimer) ... True

if (!timer instanceof ClockTimer) ... False

if (!clockTimer instanceof Timer) ... True

timer = dayTimer;

if (!timer instanceof DayTimer) ... True

if (!timer instanceof ClockTimer) ... True

Typecast

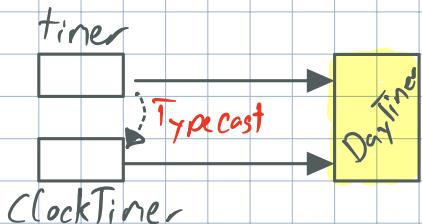
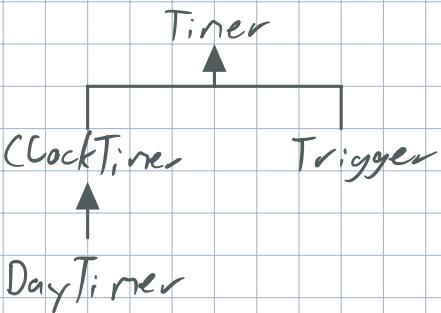
timer = new DayTimer(...);

...

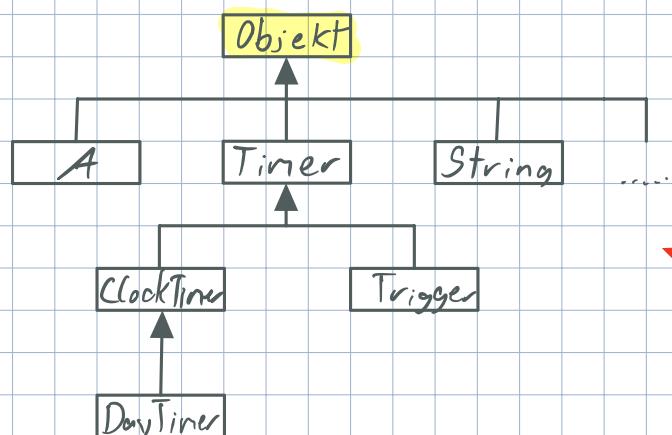
clockTimer = (ClockTimer) timer; ✓

clockTimer = (DayTimer) timer; ✓

Trigger trigger = (Trigger) timer; ↗ ClassCastException



Klasse Object



public class Object {

public
public
public
public
public final
public final
public final
...
protected
protected

Object()
String
boolean
int
Class
void
void
Object()
toString()
equals(Object obj)
hashCode()
getClass()
wait()
notify();

clone(); → Kopie
finalize(); !!! BÖSE!

Alle Klassen sind direkt oder
indirekt von Object abgeleitet

Klassendiagramm

Object x = 13;

Boxing Datentypen

new Integer();

↳ boxing

über schreiben

} 4. Klasse

Alle Klassen
sind mit Object
kompatibel

Wenn 2 Obj. mit .equals() gleich sind → selber HashCode
Wenn 2 Obj. gleichen Hashcode → nicht automatisch gleiche Obj.

Final Klassen

```
final class Account {  
    int balance;  
    void deposit(int value) {  
        balance += value;  
    }  
}
```

→ Von den final Klassen kann nicht geerbt werden.

```
void withdraw(int value) {  
    balance -= value;  
}  
}
```

Final Methoden

```
public class Account {  
    int balance;  
    final void deposit(int value) {  
        balance += value;  
    }  
}
```

⇒ Kann nicht mehr überschrieben werden

```
final void withdraw(int value) {  
    balance -= value;  
}
```

```
void print() {  
    System.out.println(balance);  
}
```

}

```

class MyAccount extends Account {
    void print() {
        System.out.println("balance = ", 00);
    }
}

```

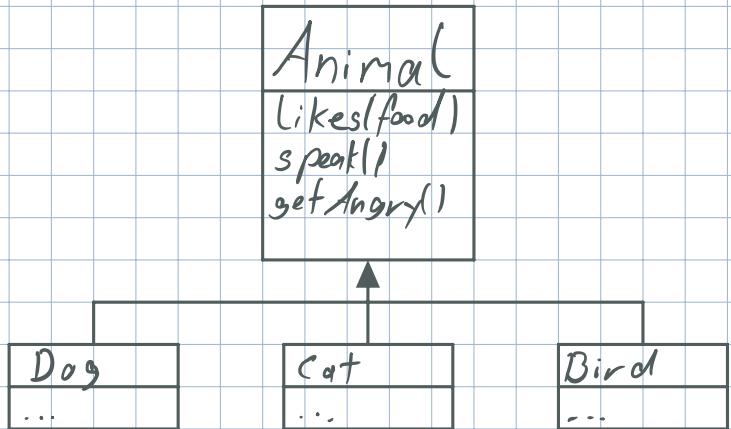
Abstrakte Klassen

```

abstract class Animal {
    String name;
    Animal(String name) {
        this.name;
    }
    void sayName() {
        System.out.println(name);
    }
}

abstract boolean likes(String food),
abstract void speak(),
abstract void getAngry();
}

```



```

class Dog extends Animal {
    Dog(String name) {
        super(name)
    }
    boolean likes(String food) {
        return "bones".equals(food);
    }
    void speak() {
        System.out.println("Wuf");
    }
    void getAngry() {
        System.out.println("Bite");
    }
}

```

Schreibweise

- Eine Methode ist abstrakt, wenn sie keine Implementierung hat; sie muss mit dem Schlüsselwort **abstract** gekennzeichnet werden.
- Eine Klasse, die mindestens 1 abstrakte Methode hat, ist selbst abstrakt und muss mit dem Schlüsselwort **abstract** gekennzeichnet werden.

Eigenschaften abstrakter Klassen

- Unterklassen müssen geerbte abstrakte Methoden implementieren (oder selbst abstrakt sein)
- Von abstrakten Klassen dürfen keine Objekte erzeugt werden:
`Animal animal = new Animal(); // Compiler meldet einen Fehler`
- Sehr wohl ist aber folgendes erlaubt:
`Animal animal = new Dog();`
`Animal[] a = new Animal[10];`

Zweck einer abstrakten Klasse

- Beschreibt eine Abstraktion (z.B. die Abstraktion **Animal**)
- Definiert Schnittstelle für zukünftige Unterklassen
 - Ist Wurzel einer Klassenfamilie
 - Ist Vorlage für weitere Unterklassen
- Programme, die mit einer abstrakten Klasse arbeiten, können auch mit allen Unterklassen davon arbeiten