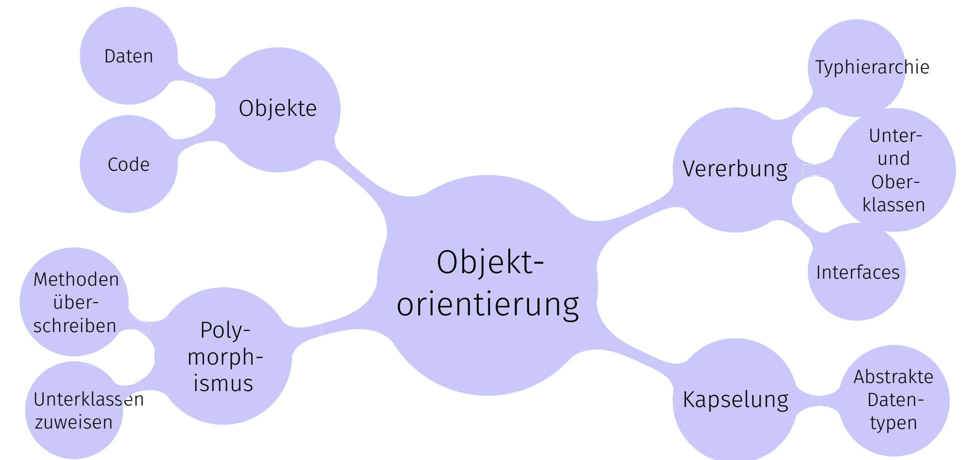
14. Java Objektorientierung

Klassen, Vererbung, Kapselung

Objektorientierung: Verschiedene Aspekte

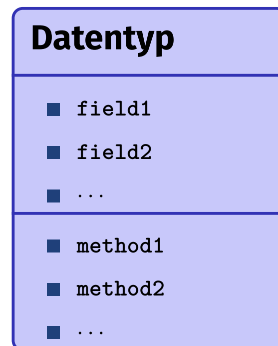


Bereits besprochen: Objekte

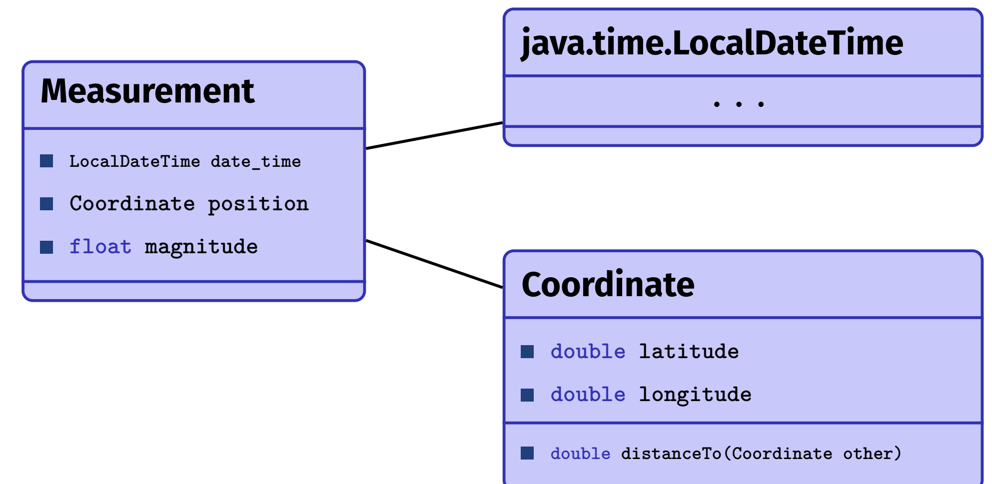
Fokus auf **Objekte** eines gegebenen Datentyps, welche

- Daten (Felder) und
- Code (Methoden)

enthalten



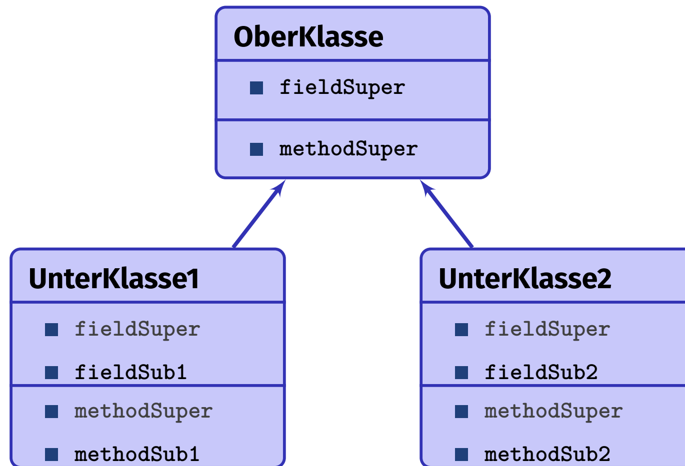
Bereits besprochen: Komposition



Vererbung

Datentypen sind in einer Typhierarchie eingegliedert.

Unterklassen (Subtypen) erben Daten und Code ihrer **Oberklassen (Supertypen)**.



Vererbung $\neq$ Komposition

Komposition: Ein Objekt enthält Felder welche Objekte von andere Typen referenzieren

Vererbung: Ein Objekt von einem Typ enthält zusätzliche Felder und Methoden, welche von einem Supertyp geerbt wurden

Korrekter Einsatz von Vererbung

Wichtige Frage bei der Überlegung, ob `DatenTyp1` von `DatenTyp2` erben soll:

Ist `DatenTyp1` ein `DatenTyp2`?

Beispiel

- **Ist** ein “Student” eine “Person” ✔
- **Ist** ein “Apfel” eine “Frucht” ✔

Korrekter Einsatz von Komposition

Wichtige Frage bei der Überlegung, ob `DatenTyp1` `DatenTyp2` als Komposition enthalten soll:

Hat `DatenTyp1` einen `DatenTyp2`?

Beispiel

- **Hat** ein “Student” eine “Address” ✔
- **Hat** ein “Apfel” eine “Farbe” ✔

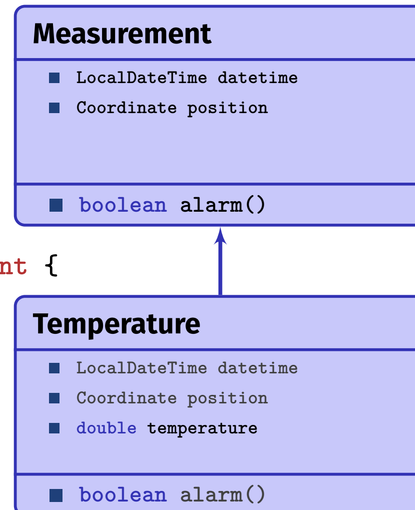
Vererbung: extends Schlüsselwort

```
class Measurement {
    LocalDateTime datetime;
    Coordinate position;

    boolean alarm() {...}
}

class Temperature extends Measurement {
    double temperature;
}

class Wind extends Measurement {
    double speed;
    double direction;
}
```

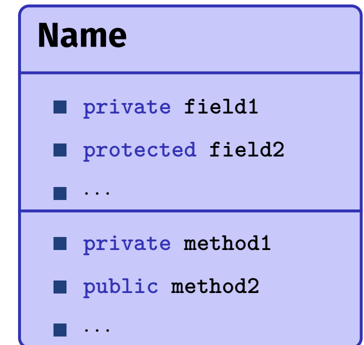


Datenkapselung (Repetition)

Steuern, welche Daten und welcher Code woher **zugänglich** ist.

Zugriffsmodifikatoren:

- private: Sichtbar aus Code derselben Klasse
- protected: Sichtbar aus Code derselben Klasse oder Unterklasse
- public: Von überall sichtbar



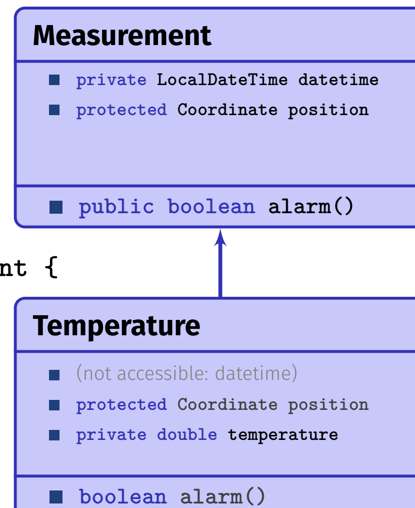
Beispiel für protected Sichtbarkeit

```
class Measurement {
    private LocalDateTime datetime;
    protected Coordinate position;

    public boolean alarm() {...}
}

class Temperature extends Measurement {
    private double temperature;
}

class Wind extends Measurement {
    private double speed;
    private double direction;
}
```



Abstrakte Klassen

```
class Measurement {
    ...
    // returns 'true' if measurement is alarming, 'false' otherwise
    public boolean alarm() {...}
}
```

- Klasse `Measurement` bietet eine Methode `alarm()` an
- Die Methode soll `true` zurückgeben, genau dann wenn die Messung **alarmierend** ist
- **... aber die implementation der Methode hängt von der implementierung der diversen Subtypen ab ... ?!**

Abstrakte Klassen

Es macht keinen Sinn, Objekte vom Typ **`Measurement`** zu erstellen. Der Datentyp sollte **abstrakt** sein.

Abstrakte Klassen: Keyword `abstract`

```
abstract class Measurement {
    ...
    // returns 'true' if measurement is alarming, 'false' otherwise
    abstract boolean alarm();
}

class Temperature extends Measurement {
    double temperature;

    // Implement the abstract method from the supertype
    boolean alarm(){
        return temperature > 35;
    }
}
```


Abstrakte Klassen: Keyword `abstract`

```
abstract class Measurement {
    ...
    // returns 'true' if measurement is alarming, 'false' otherwise
    abstract boolean alarm();
}

class Wind extends Measurement {
    double speed;

    // Implement the abstract method from the supertype
    boolean alarm(){
        return speed > 80;
    }
}
```


Abstrakte Klassen: Eigenschaften

- Falls mindestens eine Methode **`abstract`** ist, d.h. nicht implementiert, muss die ganze Klasse **`abstract`** deklariert sein.
- Abstrakte Klassen können **nicht** instanziert werden (**`new`** `...`)
- Abstrakte Klassen enhalten Daten und Code, welche von allen Subklassen geerbt wird. Von den Unterschieden wird abstrahiert.

Abstrakte Klassen: Benutzung

```
Temperature t = new Temperature(40);
boolean b = t.alarm();
```

⇒ In diesem Beispiel wird die Variable `b` auf `true` gesetzt.

Was wenn wir `alarm()` **aus einer Methode definiert in Klasse** `Measurement` **aufrufen?**

Dynamische Methodenbindung

```
abstract class Measurement {
    abstract boolean alarm();

    String alarmOutput(){
        if (this.alarm()){
            Out.println("Alarm!");
        } else {
            Out.println("Nominal");
        }
    }
}
```


Dynamische Methodenbindung

```
Temperature t = new Temperature(40);
t.alarmOutput();
```

⇒ Ausgabe: `"Alarm!"`

- Das Objekt `t` vom Typ `Temperature` erbt Methode `alarmOutput`.
- In diesem Objekt ist die Implementierung der Methode `alarm()` aus Klasse `Temperatur` an die abstrakte Methode `alarm()` gebunden.
- Deshalb wird `alarmOutput()` die Implementierung von `alarm()` aus Klasse `Temperature` aufrufen.