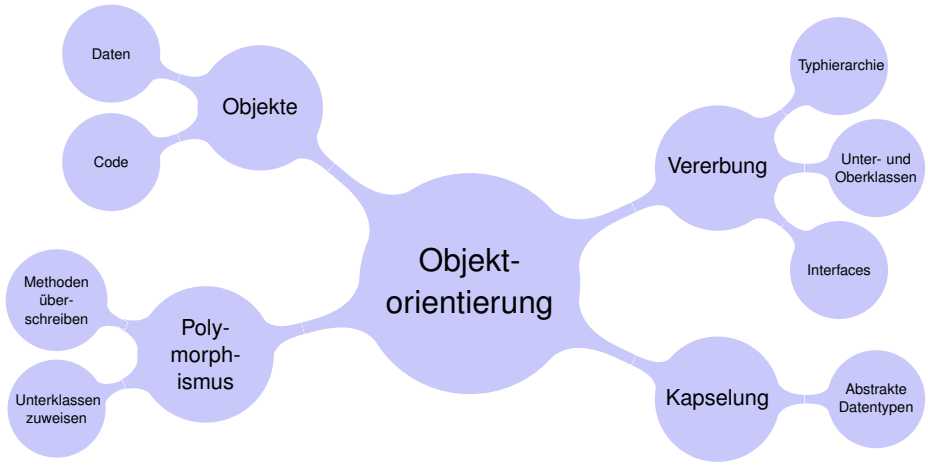
15. Java Objektorientierung II

Polymorphie

Objektorientierung: Verschiedene Aspekte



Polymorphie

- **Methoden überschreiben**: Geerbte Methoden einer Oberklasse können in der Unterklasse überschrieben werden: Gleiche Signatur, neuer Code.
- **Variablenzuweisung**: Objekte eines gegebenen Typs können Variablen eines beliebigen Supertypen zugewiesen werden.

Methoden überschreiben

Geerbte Methoden einer Oberklasse können eine neue Implementierung erhalten. Gleiche *Signatur*, neuer *Code*.

Wir erinnern uns an die Methode `alarm()` vom letzten Mal. Diese abstrakte Methode wurde in der Klasse `Wind` definiert wie folgt

```
class Wind extends Measurement {
    int speed;
    ...
    boolean alarm(){    // implements abstract method alarm()
        return this.speed > 80;
    }
}
```

Methoden überschreiben

Wir definieren nun erneut eine Unterklasse `WindWithGusts`, welche zusätzlich zur Windgeschwindigkeit und Richtung auch noch *Windböen* erfasst.

```
/*
 * Fancy windsensor data that also tracks gusts. Requires special hardware.
 */
class WindWithGusts extends Wind {
    int gusts;

    @Override
    boolean alarm(){    // replaces implementation of supertype
        return this.speed > 80 || this.gusts > 20;
    }
```


Zugriff auf überschriebene Methode: `super`

Eine Unterklasse muss beim Überschreiben einer Methode nicht den Code der Oberklasse wiederholen.

⇒ Aufruf der überschriebenen Implementation mittels dem Schlüsselwort `super`, *aber nur innerhalb der überschreibenden Implementation!*

```
class WindWithGusts extends Wind {
    int gusts;

    @Override
    boolean alarm(){    // replaces implementation of supertype
        return super.alarm() || this.gusts > 20;
    }
}
```

Führt aus: `this.speed > 80 ;`

Zugriff auf Konstruktoren der Oberklasse

Setting: Erstellen eines Messwertes erfordert immer eine Koordinate.
→ Konstruktor in Klasse `Measurement`

```
class Measurement {
    Coordinate position;

    Measurement(float lat, float lon){
        this.position = new Coordinate(lat, lon);
    }
    ...
}
```


Zugriff auf Konstruktoren der Oberklasse

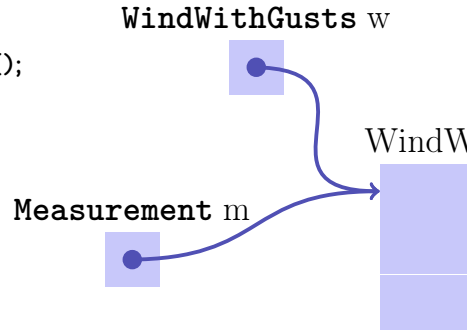
- Mit dem Schlüsselwort `super` kann ein Konstruktor einer Oberklasse aufgerufen werden.
- Die Anzahl und Typen der Argumente bestimmt, *welcher* Konstrutkur aufgerufen wird
- Der Aufruf von `super(...)` muss *immer* als erstes geschehen!

```
class Wind {
    ...
    Wind(float lat, float lon, int speed, int direction ){
        super(lat, lon);
        this.speed = speed;
        this.direction = direction;
    }
    ...
}
```

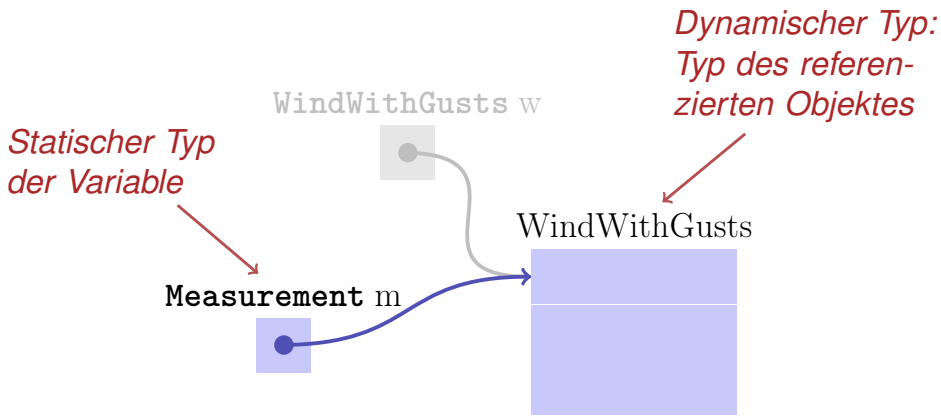
Polymorphe Referenzen

Variablen eines deklarierten Typs können Objekte eines Subtypen (Unterklasse) referenzieren, aber nicht umgekehrt.

```
WindWithGusts w = new WindWithGusts();

Measurement m;

m = w; // polymorphic reference!


// But this doesn't compile: w = m;
```



Statischer versus Dynamischer Typ

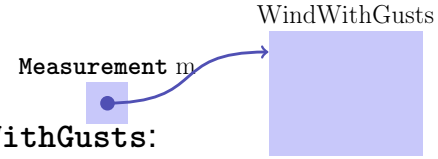


Dynamische Methodenbindung

Beim Aufruf einer Methode wird immer die Methodenimplementation des *dynamischen Typs* ausgeführt!

Aufruf:

```
m.alarm();
```



⇒ Ausgeführter Code aus Klasse `WindWithGusts`:

```
@Override
boolean alarm(){
    return super.alarm() || this.gusts > 20;
}
```


Nutzen der Dynamischen Bindung

Gegeben: Eine Liste von diversen Messwerten (*Temperaturen, Wind, ...*).

Gesucht: Eine Liste mit allen Messwerten die einen Alarm verursachen.

```
void filterByAlarm(Measurement[] measurements){
    for (int i = 0; i < measurements.length; ++i){
        if (measurements[i].alarm()){      //dynamic method binding!
            measurements[i]=null;          //remove from array
        }
    }
}
```