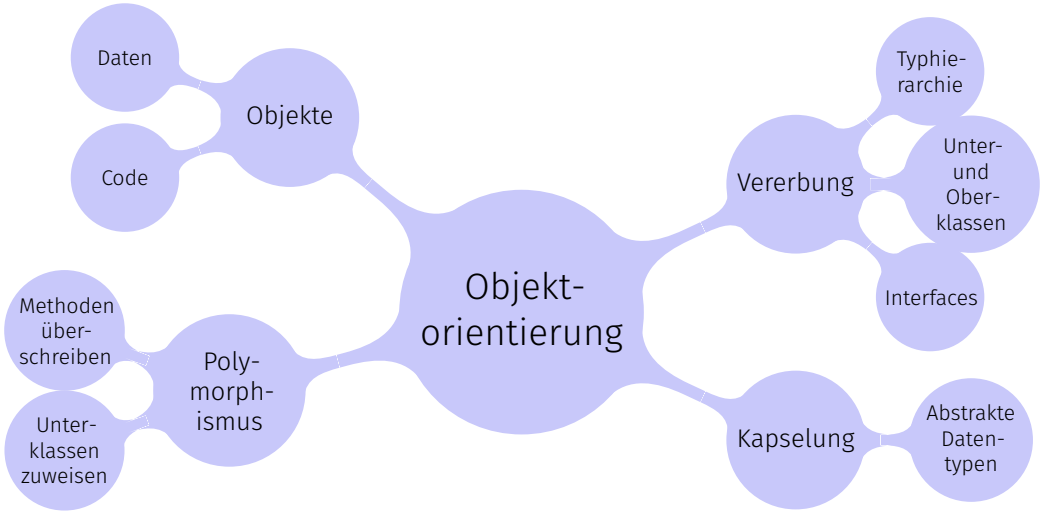
15. Java Objektorientierung II

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 that 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;
    }
}
```

Geerbt von **`Wind`**

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;
    }
}
```

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

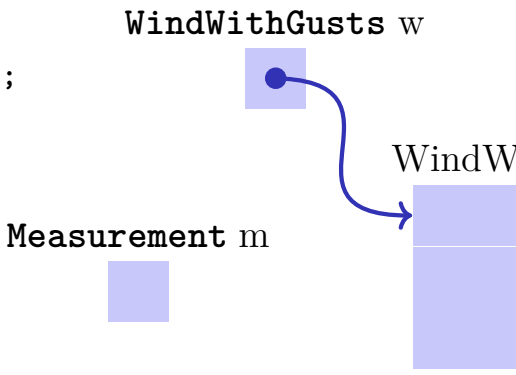
Variablen eines deklarierten Typs können Objekte eines Subtypen (Unterklasse) referenzieren, aber nicht umgekehrt.

```
WindWithGusts w = new WindWithGusts();

Measurement m;

m = w; // polymorphic reference!



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

Polymorphe Referenzen

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

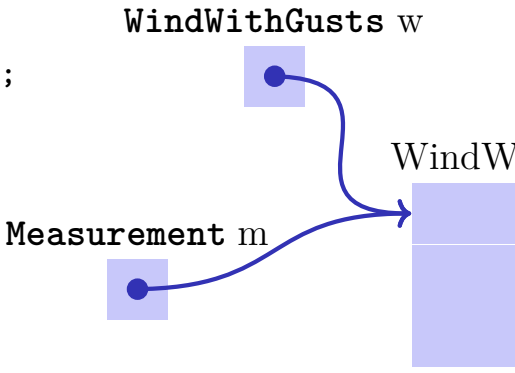
```
WindWithGusts w = new WindWithGusts();

Measurement m;

m = w; // polymorphic reference!



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



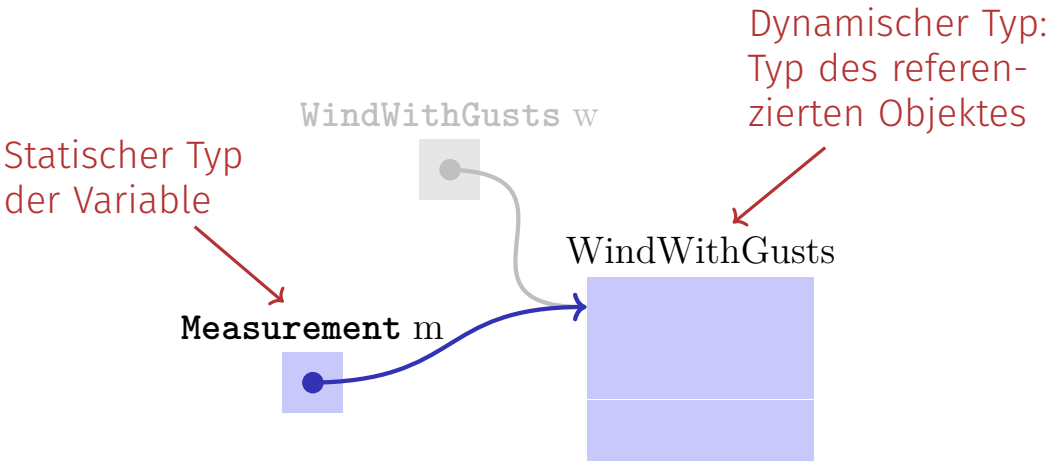
Polymorphe Referenzen

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

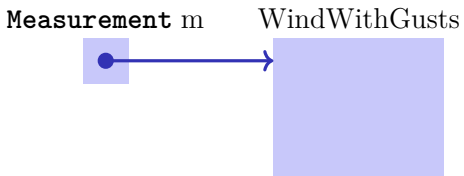


Dynamische Methodenbindung

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

Aufruf:

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

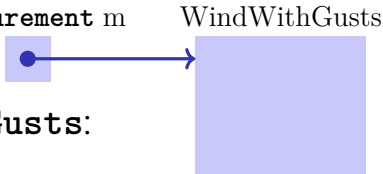


Dynamische Methodenbindung

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.

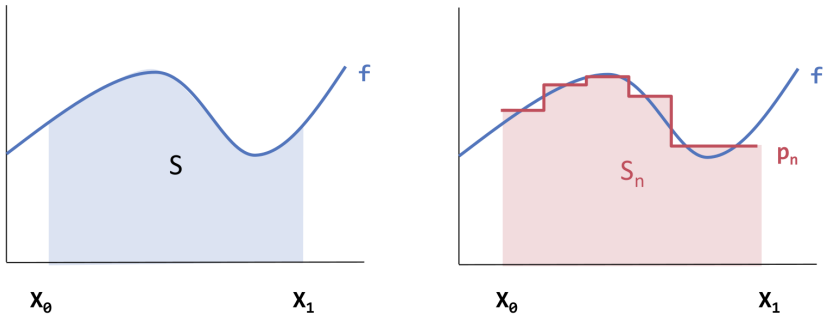
```
void filterByAlarm(Measurement[] measurements){
    for (int i = 0; i < measurements.length; ++i){
        if (measurements[i].alarm()){   //dynamic method binding!
            measurements[i]=null;       //remove from array
        }
    }
}
```

Zweites Beispiel: Numerische Integration

Ziel: Erstellen eines Softwareframeworks zur numerischen Integration einer beliebigen Funktion $f : \mathbb{R} \to \mathbb{R}$
Problem: Wie stellt man die Funktion generisch dar? (In Java gibt es keine Variablen vom Funktionstyp)
Antwort: wir verwenden Vererbung und Polymorphie.

Numerische Integration

Näherungsweise Berechnung des Integrals einer Funktion f im gewünschten Integrationsinterval $[x_0, x_1]$
Einfachster Ansatz: approximiere f durch eine stückweise konstante Funktion p_n mit n Stücken



Integriere $x \mapsto x^2$

```
public double Integrate(double x0, double x1, int n){
  double sum = 0;
  double width = (x1-x0)/n; // interval width
  for (int i = 0; i<n; ++i){
    double x = x0 + i*width + 0.5*width; // mid of interval
    double y = x*x; // function value
    sum += y * width; // rectangle area
  }
  return sum;
}
```

$$\int_{x_0}^{x_1} x^2 \, \mathrm{d}x$$

Funktionen

Generisch

```
public abstract class Function {
  public abstract double evaluate(double x);
}
```

Funktionen

Generisch

```
public abstract class Function {
  public abstract double evaluate(double x);
}
```

$x \mapsto x^2$

```
public class Square extends Function {
  @Override
  public double evaluate(double x){
    return x*x;
  }
}
```

Dichte der Normalverteilung

$$x \mapsto \frac{1}{\sqrt{2\pi}\sigma} \mathrm{e}^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

```
class Normal extends Function{
  double mu;
  double sigma;
  Normal(double m, double s)
    mu = m; sigma = s;
  }

  @Override
  public double evaluate(double x) {
    return 1/Math.sqrt(2*Math.PI)/sigma
        * Math.exp(-(x-mu)*(x-mu)/(2*sigma*sigma));
  }
}
```

Integriere (generisch)

```
public double Integrate(Function f, double x0, double x1, int n){
  double sum = 0;
  double width = (x1-x0)/n;
  for (int i = 0; i<n; ++i){
    double x = x_0 + i*width + 0.5*width;
    double y = f.evaluate(x); // evaluate of the dynamic type
    sum += y * width;
  }
  return sum;
}
```

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}x$$

Anwendung

```
Function sq = new Square();
Out.println(Integrate(sq,0,2,1000); // 2.667

Function normal = new Normal(0,1);
Out.println(Integrate(normal,-3,3,1000); // 0.997
```

Zusammenfassung der Konzepte

der objektorientierten Programmierung

Kapselung, Information Hiding

- Verbergen des Zustands und der Implementierungsdetails von Objekten
- Definition einer Schnittstelle zum Zugriff auf interne Datenstruktur $\rightarrow$ Abstraktion
- Ermöglicht das Sicherstellen von Invarianten

Zusammenfassung der Konzepte

der objektorientierten Programmierung

Vererbung

- Objekte können Eigenschaften von Objekten erben
- Abgeleitete Objekte können neue Eigenschaften besitzen oder vorhandene überschreiben
- Macht Code- und Datenwiederverwendung möglich

Zusammenfassung der Konzepte

der objektorientierten Programmierung

Polymorphie

- Ein Bezeichner kann abhängig von seiner Verwendung unterschiedliche Datentypen annehmen
- Unterschiedliche Datentypen können bei gleichem Zugriff auf ihr gemeinsames Interface verschieden reagieren
- Macht „nicht invasive” Erweiterung von Bibliotheken möglich