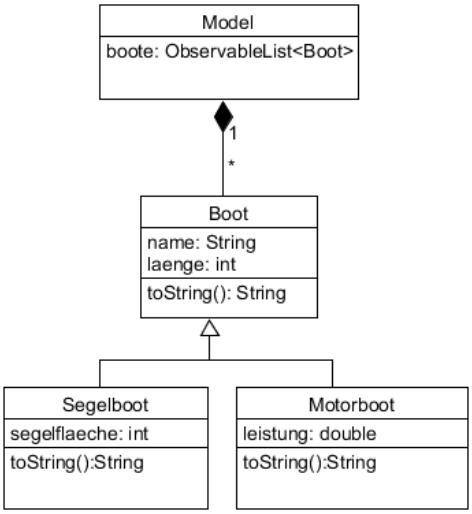
Vererbung / Polymorphismus

Ziel:

- Sie verstehen die verschiedenen Arten von Polymorphismus.
- Sie können Polymorphismus gezielt anwenden.

Ich bin auch ein Boot



Ein Segelboot ist auch ein Boot.

Ein Motorboot ist auch ein Boot.

Im Model ist ein Liste von Booten.
In dieser Liste können Objekte vom Typ ***Boot*** aber auch Objekte vom Typ ***Segelboot*** oder ***Motorboot*** sein.

Jede Klasse definiert Ihre eigene Methode toString(), die die spezifischen Attribute der Klasse ausgibt.

Dank dem **Polymorphismus** wird immer die zur Klasse passende Methode **toString()** aufgerufen, auch dann, wenn man ein Objekt als Boot anspricht.

Steckt hinter dem Boot ein Segelboot, so wird toString von Segelboot aufgerufen.

```
@Override
public String toString() {
    return "Segelboot [segelflaeche=" + segelflaeche + ", " + super.toString() + "]";
}

Beispiel einer Ausgabe
Segelboot [segelflaeche=80, Boot [name=Moeve, laenge=25]]
```

Das Sie in der abgeleiteten Klasse eine Methode erneut spezifisch implementieren können, nennt man ***Überschreiben von Methoden***.

Die **Annotierung *@Override*** ermöglicht es dem **Compiler** zu **überprüfen**, **ob wirklich eine Methode** aus der Basisklasse **überschrieben wird**.
Ob also in der Basisklasse bereits eine Methode mit einer gleichen Signatur existiert.
Falls nicht, funktioniert der Polymorphismus z.B. von t**oString** zwischen *Segelboot* und *Boot* nicht. 😉

Gleiche Signatur bedeutet: **Methoden mit gleichen Namen und Parametern**

Polymorphismus Unterscheidungen

Quelle: www.programmieraufgaben.ch „Objekte und Klassen" oo.pdf

Polymorphie von Methoden in **nicht voneinander abgeleiteten Klassen**:

- In Klassen (die nicht voneinander vererbt sind) können Methoden mit gleichen Namen und Parametern (gleiche Signatur) definiert sein.
- **Diese Methoden** haben ausser der Signatur **keine Gemeinsamkeiten**.

```
public class Ausleihe {

    public boolean beenden(Date datum){
        // Buch zuruckgeben
    }
}

public class Arbeitsverhaeltnis {

    public boolean beenden(Date datum){
        // Mitarbeiter verlasst Firma
    }
}
```

Statischer Polymorphismus (Methoden mit unterschiedlichen Signaturen)

- In einer Klasse können mehrere Methoden gleich heissen, jedoch **unterschiedliche Parameter** haben. ➔ Also eine **unterschiedliche Signatur**.
- Ein Bespiel ist dazu die *System.out.println* Methode, die es mit den Parametern *boolean, char, int, long, Object* und *String* gibt.
- Die Methode macht immer das Gleiche, verwendet jedoch anderer Typen beim Parameter.

```
public class ConsoleOutput {

    public void println(boolean value){
        // boolean und \n ausgeben
    }

    public void println(int value){
        // int und \n ausgeben
    }

    public void println(long value){
        // long und \n ausgeben
    }
}
```

Erst im folgenden dritten Beispiel kommt der **Vorteil der Vererbung** zum Tragen.

Dynamischer Polymorphismus

- Gehen wir davon aus, dass von der **Basis-Klasse *Person*** die Klasse *Student* abgeleitet wurde.
  Student ist also **eine abgeleitete Klasse**, Kind Klasse.
- Der **dynamische Polymorphismus erlaubt** es, dass **einer Variablen vom Typ *Person*** problemlos **ein Objekt der Klasse *Student* zugewiesen werden kann**.
- Die **dynamische Bindung** ist ein Begriff, der den Umgang des Compilers mit polymorphen Methoden beschreibt. Man spricht von **dynamischer Bindung**, wenn ein **Methodenaufruf zur Laufzeit** anhand des **tatsächlichen Typs eines Objektes** aufgelöst wird.
  Beispiel:
  Eine Liste von Personen umfasst auch Studenten oder Ehemalige.
  Ruft man für jedes Element der Liste die Methode getValue() auf, so wird dynamisch anhand des tatsächlichen Typs des Objektes die "spezialisierte" Methode getValue() aufgerufen. Für eine Person, die effektiv ein Student ist, also getValue() der Klasse Student.

```
public class Person{
    private String name;

    //kein @Override wenn Super-Klasse
    public int getValue(){
        //...
    }
}

public class Student extends Person{
    private Date studiumsstart;

    @Override
    public int getValue(){
        //...
    }}

public class Ehemaliger extends Student{
    private Date abschluss;

    @Override
    public int getValue(){
        //...
    }
}


Ehemaliger albertE = new Ehemaliger();
Student paulM = new Student();

Student student01 = albertE;
Person  person02  = albertE;
Person  person03  = paulM;

student01.getValue(); //ruft Ehemaliger:getValue() auf

//Sowohl Studenten wie Ehemalige können zusammen
//in einer Liste<Person> verwaltet werden.
ObservableList <Person> personen;
personen.add(albertE);
personen.add(paulM;
```