PR2 – Formular für Lesenotizen

SS2021

Nachname	Vorname	Matrikelnummer	Abgabedatum:
Lushaj	Detijon	1630149	22.04.21

Eigene Exception-Klassen

- Man kann von jeder beliebigen Exception und auch von Throwable und Error erben.

```
public class RadiusException extends ArithmeticException {
    private int radius;
    public RadiusException(int radius, String msg) {
        super(msg);
        this.radius= radius;
    }
    public int getRadius() {
        return radius;
    }
}
```

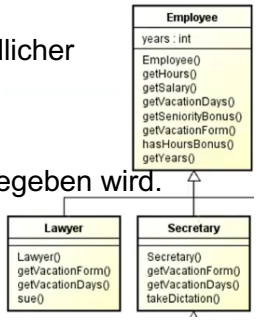
Klasse mit Objektzähler

```
public class Spieler {
    private static int lastnum = 0;
    public Spieler(String name) {
        lastnum++;
        if (lastnum < 12) {
            this.name = name;
            this.num = lastnum;
        } else {
            throw new IllegalStateException("…");
        }
    }
    @Override public String toString() {
        return name + " (" + num +")";
    }
}
```

L.6 Polymorphie

Polymorphie / Polymorphismus: Programmcode kann unverändert für verschiedene Objekttypen eingesetzt werden. Das Programm verhält sich dabei jeweils unterschiedlich.

"**Polymorphismus** heißt, dass gleich lautende Nachrichten an kompatible Objekte unterschiedlicher Klassen ein unterschiedliches Verhalten bewirken können. `-dynamische Binden`



6.2 Statischer und dynamischer Typ

Statischer Typ: Objekttyp, der bei Deklaration einer Variablen als ihr Typ (links vom Namen) angegeben wird.

- Bestimmt, welche Methoden überhaupt aufgerufen werden dürfen

Dynamischer Typ: Objekttyp hinter `new` bei Konstruktor-Aufruf.

- Bestimmt, welche (evtl. überschriebene) Methode ausgeführt wird

```
Employee lisa= new Lawyer(5) //dynamischer Typ: Lawyer || statischer Typ Employee
int hours = lisa.getHours(); // ok
lisa.sue(); // compiler error            da .sue eine methode von Lawyer ist und nicht von Employee
```

Typumwandlung (type cast)

Downcast: Typumwandlung einer Variable in einen Subtyp

```
Employee lisa = new Lawyer(5);
Lawyer theRealLisa= (Lawyer)lisa;
theRealLisa.sue(); // ok
```

Upcast: Typumwandlung einer Variable in einen Supertyp (i. d. R. implizit)

```
Lawyer linda = new Lawyer(0);  // implicit upcast:
linda.getHours(); //getHours ist geerbte Employee-Methode
((Employee)linda).getHours(); //explizite Variante
```

Das geht nur entlang der Vererbungshierarchie || Keine Typ-Umwandlung zu "Geschwistern"

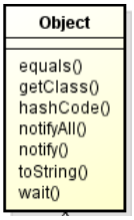
Typisierungen von Programmiersprachen

statisch typisierte Typprüfung vom Compiler
- Fehlerhafte Operationen werden früh erkannt
- Seltene böse Überraschungen zur Laufzeit

dynamisch typisierte Typprüfung zur Laufzeit
- Höhere Flexibilität
- Keine Notwendigkeit, die statischen Typprüfungen mühsam zu umgehen

L.6.3 Die Klasse Object

Alle Objekttypen haben eine Superklasse `Object`, und sie erben ohne `extends`-Angabe implizit von Object.



Das Schlüsselwort instanceof

- Ermöglicht Abfrage, ob eine Variable auf ein Objekt eines gegebenen Typs referenziert.
- Subklassen des Typ sind auch true

```
if (variable instanceof type) {
    statement(s);
}
```

equals und hashCode

Die Methode hashCode liefert einen möglichst eindeutigen Wert in Form eines ints zur Identifikation des Inhalts eines Objekts zurück

```
java.util.Objects.hash(variable, y);
```

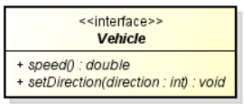
```
@Override public boolean equals(Object o) {
    if (o!=null && o.getClass()==getClass()) { /V2
    if (o instanceof Loc) {
        Loc loc = (Loc) o; // cast and compare it
        return (x == loc.x && y == loc.y);
    } else {
        return false; // o is not a Loc; cannot be equal}}

@Override public int hashCode() {
    return java.util.Objects.hash(x, y);
    return name.hashCode() + new
Boolean(istBundeseinheitlich).hashCode();
}
```

L.6.4 Interfaces

Interface: Eine Liste von Methodenköpfen, die eine Klasse implementieren kann. -dynamisch gebunden

- **Implementierungsvererbung** definiert eine "ist-ein"-Beziehung und vererbt dabei eine Signatur mit Implementierung an die Subklasse.
- **Interfaces** definieren eine "ist-ein"-Beziehung ohne Implementierung(svererbung): Nur Signatur ist vorhanden und wird vererbt




```
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...}


public class name implements interface {
    ...
}
```

```
public interface HasArea {
    public double area();
}
public interface Shape implements HasArea {
    public double perimeter();
}

public class Rectangle implements Shape {
    ...
    @Override public double area() {
        return width * height;    }
    @Override public double perimeter() {
        return 2.0 * (width + height);
    } }
```

→ **Abstrakte Methode**: Ein Methodenkopf ohne Implementierung.

- Ein Interface zwingt zur Implementierung → Compilerfehler
- Alle Methoden eines Interface sind implizit public.
- Interfaces können Klassenkonstanten definieren (sind immer `public static final`).
- Interfaces können statische Methoden nicht vererben.
- Klassen können mehrere Interfaces gleichzeitig implementieren.

L. 6.5 Substitutionsprinzip

Liskovsche Substitutionsprinzip

Ein für Clients relevantes Versprechen einer Superklasse (oder eines Interfaces) soll auch von allen Subklassen eingehalten werden.

Substitutionsprinzip:

- Jedes Objekt des Subtyps kann aus Sicht eines Clients ein Objekt des Supertyps ersetzen.
- Jede für den Client relevante Eigenschaft eines Objekts des Supertyps muss auch für ein beliebiges Objekt des Subtyps gelten.

Beispiel – Rechteck und Quadrat

```
public class Rectangle implements Shape {
    ...
    public void stretch(double factorW, double
factorH) {
        width *= factorW;
        height *= factorH;
    }
}
```

```
public class Square extends Rectangle {
    ...
    @Override public void stretch(double
factorW, double factorH){
        super.stretch(factorW, factorW);
    }
}
```

→ Damit ist das Substitutionsprinzip verletzt.

- Das Substitutionsprinzip ist also abhängig vom jeweiligen Kontext.
  - Für unveränderliche Rechtecke folgt eine Subklasse Quadrat dem Substitutionsprinzip.
  - Für veränderliche Rechtecke – wie wir oben gesehen haben – nicht.
  - Ob das Substitutionsprinzip gilt, hängt also davon ab, was der Client-Code mit dem Objekt macht ("was für den Client relevant ist")
- Definieren Sie nur dann eine Vererbungsbeziehung, wenn das Substitutionsprinzip gilt.
- Achten Sie bei Erweiterungen der beteiligten Klassen darauf, dass das Substitutionsprinzip weiterhin gilt.

→ Generell möchte ich die Empfehlung geben, Vererbung als Mittel der Wiederverwendung eines Objektzustandes sehr dosiert einzusetzen.