Vererbung und Polymorphie

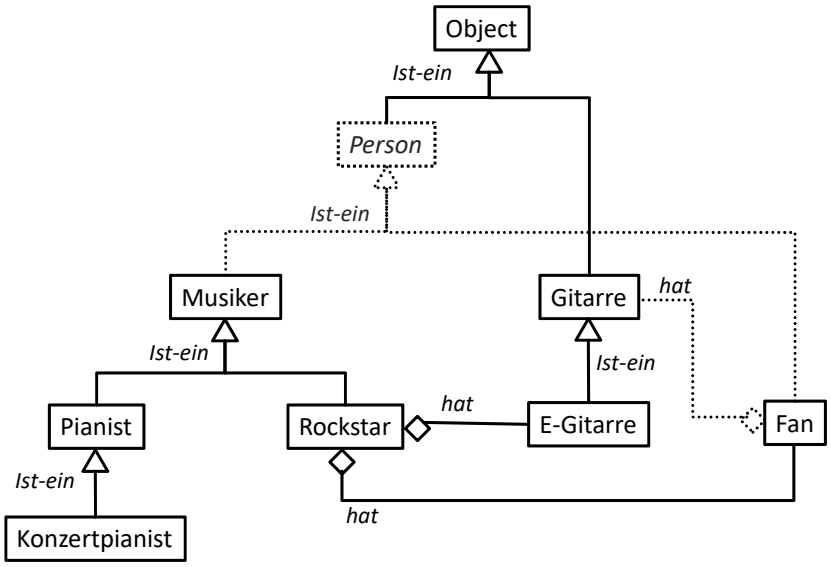
Aufgabe 1: Vererbungsbaum entwerfen

a) Finden Sie sinnvolle Beziehungen. Füllen Sie die letzten Spalten aus.

Hinweis: Es kann nicht alles mit etwas anderem verbunden werden.

Klasse	**Ist-Ein (Is-a)**	**Hat-Ein (Has-a)**	**Oberklasse**	**Unterklassen**
Musiker	---		*Optional: Person* (bzw. Object)	Pianist, Rockstar
Rock-Star	Musiker	Fan, E-Gitarre	Musiker	---
Fan	---	*Optional: Gitarre*	*Optional: Person* (bzw. Object)	---
Gitarre	---		--- (Object)	E-Gitarre
Pianist	Musiker	---	Musiker	Konzertpianist
E-Gitarre	Gitarre	---	Gitarre	---
Konzertpianist	Pianist	---	Pianist	---

b) Zeichnen Sie die Vererbungsklassendiagramme. Gibt es ggf. auch Hat-Ein-Beziehungen, welche Sie einzeichnen können?



Diese Seite ist leer, Aufgabe 2 siehe nachfolgende Seite.

Aufgabe 2: Überschreiben und Überladen von Methoden

```
class A {

      void f() { System.out.println("A.f()"); }

      void f(int i) { System.out.println("A.f(int)"); }

      void g() { System.out.println("A.g()"); }

}

class B extends A {

      void f() { super.f();
                 System.out.println("B.f()");
      }

      void g(int i) { System.out.println("B.g(int)"); }

      void h() { System.out.println("B.h()");
                 this.g();
      }
}
```

a) `A` ist Oberklasse der Klasse `B`. Welche Methoden besitzt die Klasse `B` (ohne die von `Object` geerbten Methoden)?
Entscheiden sie für die Methoden der Klasse `B`, ob sie jeweils geerbt, überschrieben oder zusätzlich hinzugekommen sind. Welche Methoden werden überladen?

Methode	Geerbt?	Überschrieben?	Zusätzlich?	Überladen?	Begr.
`f()`	**Jein:** stammt aus A (aber über-schrieben)	**Ja:** Code in B geändert	**Nein:** gab es schon in A	**Ja:** es gibt auch noch `f(int)`	
`f(int)`	**Ja**	**Nein:** keine Änderung in B	**Nein**	**Ja:** es gibt auch noch `f()`	
`g()`	**Ja**	**Nein**	**Nein**	**Ja:** es gibt auch noch `g(int)`	
`g(int)`	**Nein:** neu in B	**Nein:** Nur bei geerbten Methoden möglich	**Ja:** gab es vorher noch nicht (in A)	**Ja:** es gibt auch noch `g()`	
`h()`	**Nein:** neu in B	**Nein:** Nur bei geerbten Methoden möglich	**Ja:** `h` gab es vorher noch nicht (in A)	**Nein:** Es gibt nur ein `h`	

b) Welche der folgenden Anweisungen sind korrekt? Was wird – falls korrekt – jeweils ausgegeben (und welche Methode wird dabei ausgeführt)?

Anweisung	**korrekt?**	**Ausgabe**	**Begründung**
`A o = new A();` `o.f();`	**Ja**	`A.f()`	
`B o = new B();` `o.f();`	**Ja**	`A.f()` `B.f()`	`f()` überschrieben: `super.f()` = Aufruf `A.f()`
`A o = new B();` `o.f();`	**Ja**	`A.f()` `B.f()`	- B ist Unterklasse von A → an A zuweisbar - Verhalten jedoch abhängig vom Objekt-Typ (B) → Polymorphie
`A o = (A) new B();` `o.f();`	**Ja**	`A.f()` `B.f()`	Wie zuvor ; Casting B → A ist hier überflüssig
`B o = new A();` `o.f();`	**Nein**		Oberklasse A kann nicht an Variable der Unterklasse zugewiesen werden (A fehlen die zusätzlichen Methoden aus B)
`B o = (B) new A();` `o.f();`	**Nein**		Für Compiler OK, aber **Laufzeit-fehler** (ClassCastException)
`A o = new A();` `o.g(5);`	**Nein**		Methode `g(int)` in A undefiniert
`B o = new B();` `o.g(5);`	**Ja**	`B.g(int)`	
`A o = new B();` `o.h();`	**Nein**		Zusätzliche Methode `h()` aus B in A nicht bekannt → kann nicht aufgerufen werden
`B o = new B();` `o.h();`	**Ja**	`B.h()`	
`B o = new B();` `o.toString();`	**Ja**	B + Adresse von o	

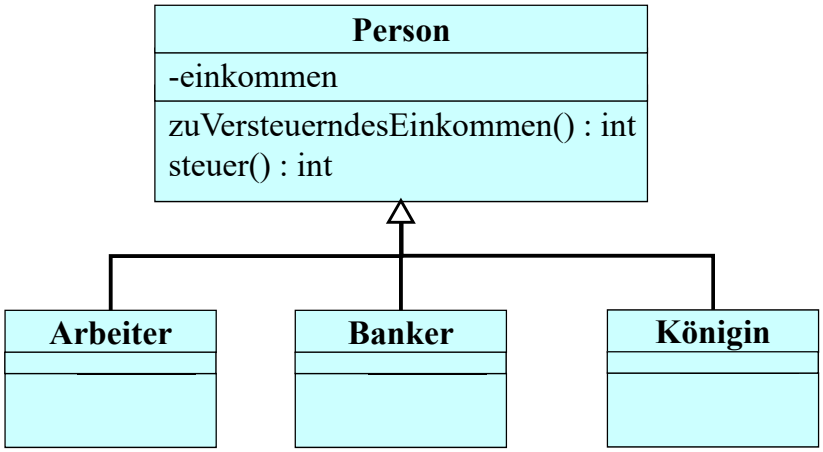
c) Übersetzen Sie die Klassen mit `javac`. Betrachten Sie mit `javap` den Inhalt der erzeugten Klassendateien A.class sowie B.class (Hinweis: Aufruf mit `javap A` bzw. `javap B`).
Achten Sie insbesondere auf die Vererbungshierarchie und die Liste der Methoden – Fällte Ihnen dabei etwas auf?

Ausgabe:

```
> javap A
      Compiled from "A.java"
      class A extends java.lang.Object{
          A();
          void f();
          void f(int);
          void g();
      }

> javap B
      Compiled from "A.java"
      class B extends A{
          B();
          void f();
          void g(int);
          void h();
      }
```

Aufgabe 3: Polymorphie

In modernen Ländern basiert das Finanz- und Steuersystem auf IT-Systemen. Die verschiedenen Personen eines Landes seien durch folgende Klassenhierarchie modelliert:



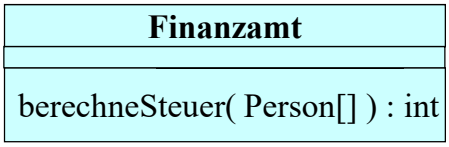
Das Attribut ***einkommen*** enthält das tatsächliche Jahreseinkommen einer Person (in Pfund £). Die Methoden ***int zuVersteuerndesEinkommen()*** und ***int steuer()***

sollen jeweils die korrekten Werte entsprechend der nachfolgenden (vereinfachten und nicht sehr realistischen ☺) Regeln berechnen:

1. Jede **Person** muss Steuern auf sein gesamtes Einkommen zahlen, sofern diese Regeln nichts anderes besagen. Folglich entspricht das zu versteuernde Einkommen normalerweise dem tatsächlichen Einkommen einer Person.
2. Jede Person muss 25% seines (zu versteuernden) Einkommens als Steuer bezahlen. Die Steuer kann dabei auf ganze Pfund abgerundet werden.
3. **Arbeiter** erhalten einen Steuerfreibetrag von 2.400 £. Dieser ist für die Kosten des Weges zur Arbeit und dergleichen gedacht.
4. Solange die Banken in der gegenwärtigen Krise von der Regierung unterstützt werden, muss jeder **Banker** 1.000 £ zusätzlich an Steuern bezahlen.
5. Die **Königin** braucht gar *keine* Steuern zu bezahlen.
6. Der zu zahlende Steuerbetrag ist niemals höher als das Einkommen einer Person, und niemals negativ.

Da mit jährlichen Änderungen des Steuersatzes zu rechnen ist, soll Ihre Implementierung entsprechend änderungsfreundlich sein. Die Steuerberechnung gemäß Regel 2 soll daher nur an einer Stelle in der Klassenhierarchie erfolgen.

Das Finanzamt kann die gesamten Steuereinnahmen des Landes mit Hilfe der Methode ***int berechneSteuer(Person[])*** berechnen:



a) Implementieren Sie die Klassen der ***Person***-Hierarchie. Welche Methoden sollten in der Klasse ***Person*** implementiert werden und welche Methoden sollten in ihren Unterklassen überschrieben werden?

b) Implementieren Sie das Finanzamt (Klasse ***Finanzamt***). Vergessen Sie nicht ihr Programm entsprechend zu testen.

c) Wie viel Steuern kann das Finanzamt von der folgenden Personen-Gruppe eintreiben?

Person	**Einkommen**
Person (“Joe Unemployed”)	6.400 £
Arbeiter (“Suzi Hard-working”)	36.000 £
Banker (“Fred Moneymaker”)	4.000.000 £
Königin (“Elisabeth”)	1.000.000 £

d) Implementieren Sie zum Testen ihrer Funktionen eine geeignete ***toString()***-Methode, um zu den Personen folgende Daten auszugeben:
   - das (Brutto-)Einkommen (vor Steuern)
   - das zu versteuernde Einkommen
   - die zu zahlende Steuer
   - das übrigbleibende Netto-Einkommen (Brutto-Einkommen abzüglich der Steuer)

e) (Optional) Erweitern Sie zur besseren Lesbarkeit die Klasse ***Person*** um einen Namen.

Lösung: Person.java

```
public class Person {
	private int einkommen;
	private String name = "";		// Optional

	public Person(int einkommen) {
		this.einkommen = einkommen;
	}
	public Person(int einkommen, String name) { // Optional
		this.einkommen = einkommen;
		this.name = name;
	}

	public int zuVersteuerndesEinkommen() {
		return einkommen;
	}

	private final double STEUER_SATZ = 0.25;

	public int steuer() {
		double steuer = Math.floor( STEUER_SATZ
					* this.zuVersteuerndesEinkommen() );
		return (int) steuer;
	}

	@Override public String toString() {
		int netto = this.einkommen - this.steuer();
		String className = this.getClass().getName();
		return className + "(" +
			this.name +		// Optional
			"): " +
			"\n\t\t Brutto-Einkommen: " + this.einkommen +
			"\n\t\t zu verst. Einkommen: " +
			    this.zuVersteuerndesEinkommen() +
			"\n\t\t Steuer: " + this.steuer() +
			"\n\t\t Netto-Einkommen: " + netto;
	}
}
```

Lösung: Arbeiter.java

```java
public class Arbeiter extends Person {

	public Arbeiter(int einkommen) {
		super(einkommen);
	}
	public Arbeiter(int einkommen, String name) { // Optional
		super(einkommen, name);
	}

	@Override public int zuVersteuerndesEinkommen() {
		int einkommen = super.zuVersteuerndesEinkommen() - 2400;
		//return Math.max(einkommen, 0);
		if (einkommen >= 0)
			return einkommen;
		else
			return 0;
	}
}
```

Lösung: Banker.java

```java
public class Banker extends Person {

	public Banker(int einkommen) {
		super(einkommen);
	}
	public Banker(int einkommen, String name) { // Optional
		super(einkommen, name);
	}

	@Override public int steuer() {
		// aufpassen: Steuer > Einkommen möglich
		//     --> max. Einkommen zurückgeben
		//return super.steuer() + 1000;
		return Math.min(super.steuer() + 1000 /* Steuer*/,
				this.zuVersteuerndesEinkommen() /* Einkommen */ );
	}


}
```

Lösung: Königin.java

```
public class Königin extends Person {

	public Königin(int einkommen) {
		super(einkommen);
	}
	public Königin(int einkommen, String name) { // Optional
		super(einkommen, name);
	}

	@Override public int steuer() {
		return 0;		// das ist sicher
	}

	@Override public int zuVersteuerndesEinkommen() {
		return 0;		// optional
	}
}
```

Lösung: Finanzamt.java

```
public class Finanzamt {

	public int berechneSteuer(Person[] einwohner) {
		int tax = 0;
		for (int i = 0; i < einwohner.length; i++) {
			tax += einwohner[i].steuer();
		}

		return tax;
	}
}
```

Lösung: TestFinanzamt.java

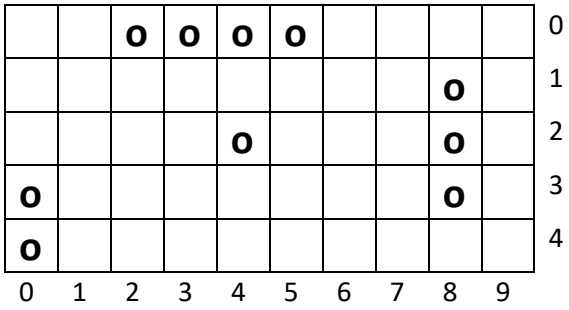
```
public class TestFinanzamt {

	public static void main(String[] args) {
		Finanzamt finanzamt = new Finanzamt();

		Person[] personen = {
				new Person(     6400 , "Joe Unemployed"),// Name:
Optional
				new Arbeiter(  36000 , "Suzi Hard-working"),
				new Banker(  4000000 , "Fred Moneymaker"),
				new Königin( 1000000 , "Elisabeth")
//				new Person(     6400 ),
//				new Arbeiter(  36000 ),
//				new Banker(  4000000 ),
//				new Königin( 1000000 )
		};

		for (int i = 0; i < personen.length; i++) {
			System.out.println("Person[" + i + "]: "
					+ personen[i]);
```

```
//                              + " zu verst. Einkommen = "
//                              + personen[i].zuVersteuerndesEinkommen()
//                              + ", Steuer = " + personen[i].steuer());
            }
            System.out.println();

            int steuerGesamt = finanzamt.berechneSteuer( personen );
            System.out.println("Gesamte Steuern = " + steuerGesamt);


            // TODO: Testfälle
            //     - Banker: Steuer (+1000) nicht größer als Einkommen
            //     - Arbeiter: zuVersteuerndesEinkommen nicht negativ
            //     - Arbeiter: Steuer 25% von (Einkommen -2400) (korrekt),
            //                         oder 25% von Einkommen (falsch --> Freibetrag
fehlt)
            //                         oder 25% Einkommen - 2400 (falsch -->
Freibetrage: 9.600 !!!)
        }
}
```

Zusatz-Aufgabe 4: Schiffe-Versenken

Realisieren Sie das Spiel Schiffe-Versenken mit einem 2-dimensionalen Spielfeld.

Erweitern Sie dazu Ihre Klasse **SchiffeVersenken1D** aus Übung 1, indem Sie ein 2-dimensionales Array von **Schiff**-Objekten (siehe Aufgabe 4 in Übung 2) anlegen. Dabei sollen mehrere Schiffe (idealerweise unterschiedlich groß und mit verschiedener Ausrichtung horizontal bzw. vertikal) platziert werden.



Hinweis:

- Bei der initialen Platzierung der Schiffe können Sie die Methode **overlapps** verwenden, um zu testen, ob ein zu platzierendes Schiff mit einem anderen kollidieren würde.
- Frage: Wie können Sie überprüfen, ob ein zufällig erzeugtes Schiffs-Objekt (d.h. mit zufällig gewählter Position, Größe & Ausrichtung in x- bzw. y-Richtung) über den Spielfeldrand ragt?

Zusatz-Aufgabe 5 (für Java-Experten): Java-Rätsel: instanceof und Cast

Was machen jeweils die folgenden Java-Programme?

Hinweis:

- Versuchen Sie zunächst, die Lösung „im Kopf“ zu finden, und überprüfen Sie erst danach Ihre Vermutung am Rechner. Versuchen Sie dann herauszufinden, warum Ihre gedachte Lösung ggf. falsch war.
- Bei Fragen wenden Sie sich direkt an den Dozenten.

```
public class Type1 {
   public static void main(String[] args) {
     String s = null;
     System.out.println(s instanceof String);
   }
}
```

Ausgabe:

```
false
```

Erklärung:

- Zwar ist **`null`** zu jedem Objekt-Typ kompatibel.
- Allerdings ist der **`instanceof`**-Operator so definiert, dass er **`false`** liefert, wenn der linke Operand null ist.
- Dieses Verhalten ist sehr sinnvoll, da dadurch nach dem Test mit instanceof das jeweilige Objekt ohne Probleme zum entsprechenden Typ gecastet werden kann ohne Gefahr einer **`ClassCastException`** oder **`NullPointerException`**.

Kurzfassung:

- `null instanceof XXX` → immer `false`
- sinnvoll, da sonst bei nachfolgendem Cast eine `NullPointerException` folgen würde

```
// ------------------------------------------------

public class Type2 {
   public static void main(String[] args) {
     System.out.println(new Type2() instanceof String);
   }
}
```

Ausgabe:

- Keine, da das Programm gar nicht kompiliert werden kann.
- Es gibt einen Compilerfehler: `Type2` ist kein Untertyp von `String` (nur dann macht instanceof Sinn wegen nachfolgendem Cast, der geht hier aber nie → deshalb bereits Compilerfehler)

```
Incompatible conditional operand types Type2 and
String
```

Erklärung:

- Der **`instanceof`**-Operator erfordert, dass falls beide Operanden Klassen sind, einer ein Untertyp des anderen sein muss.
- Offensichtlich ist **`Type2`** kein Untertyp von **`String`**, folglich kann das Objekt nicht in ein String gecastet werden.
- Dies kann bereits der Compiler feststellen und „meckert“, sodass es gar nicht erst zur Laufzeit zu einem Problem kommen kann.

```
// ------------------------------------------------

public class Type3 {
   public static void main(String[] args) {
     Type3 t3 = (Type3) new Object();
   }
}
```

Ergebnis:

- Laufzeitfehler: `ClassCastException`

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.Object cannot be cast to Type3
          at Type3.main(Type3.java:4)
```

Erklärung:

- **`Object`** kann nicht zu **`Type3`** gecastet werden. (Offensichtlich ist **`Type3`** kein Obertyp von **`Object`** – hier ist es genau anders herum.)
- Das Typsystem von Java ist nicht mächtig genug, um dies bereits zur Compilezeit zu erkennen.
- Allerdings schützt uns die Java Virtual Machine (JVM), da bei der Typumwandlung (Casten) jeweils zur Laufzeit geprüft wird, um die Typumwandlung überhaupt möglich ist. Im Fehlerfall wird eine **`ClassCastException`** ausgelöst.