NACHNAME:	`Gonal`	**SEMESTER:**	☐ **M5** ☐ **M6** ☐ M3 ☐ M4 ☐ M7
VORNAME:	`Poly`	**VERTIEFUNG:**	☐ **FV** ☐ **IM**

VERWENDETE KLASSEN:

- Als **Anlage** erhalten Sie den Ausdruck des vorab bekannt gemachten Quelltextes von vier Versionen einer Klasse `Polygon`. Ein `Polygon` soll (in allen vier Versionen) ein ebenes Polygon aus zwei oder mehr Punkten beschreiben.
- Darüber hinaus wird die aus der Studienarbeit bekannte Klasse `Punkt` verwendet mit der einzigen Änderung, dass die Klasse **nicht** `final` ist. Die Klasse wird als bekannt vorausgesetzt.

UNBEDINGT BEACHTEN:

- **Bevor** Sie mit der Bearbeitung beginnen, müssen die Angaben zur Person auf dieser Seite **vollständig** ausgefüllt sein.
- Es sind **keinerlei Hilfsmittel** zugelassen.

GENERELLE VORGABEN:

- Es sind keinerlei Kommentare verlangt, weder `javadoc`-Kommentare noch andere.
- Programmier-Richtlinien (insbesondere Checkstyle) sind zu beachten. Bei Testklassen dürfen aber *magic numbers* verwendet werden.

Aufgabe 1: (30 Punkte)

Eine Testklasse `PolygonTest` beginne wie folgt:

```
public final class PolygonTest {
    private static final Punkt A1 = new Punkt(1, 0);
    private static final Punkt A2 = new Punkt(1, 0);
    private static final Punkt B = new Punkt(1, 1);
    private static final Polygon P = new Polygon(A1, B);
    private static final Polygon Q = new Polygon(A1, B);
    private static final Polygon R = new Polygon(A1, B, A2);
    private static final Polygon S = new Polygon(A2, B);
    private static final Polygon T = new Polygon(A2, A2);
```

a) Prüfen Sie, ob die in der linken Spalte angegebenen **einzelnen** Tests jeweils gut gehen. Tragen Sie in diesem Falle **OK** ein, anderenfalls **F** (für Failure). (Ein Error entsteht nirgends.) Prüfen Sie bei jeder Version **alle** Tests, auch wenn schon eine Failure erkannt wurde.

	v1	v2	v3	v4
`assertTrue(P.equals(Q));`	**OK**	**OK**	**OK**	**OK**
`assertTrue(S.equals(P));`	**F**	**OK**	**OK**	**OK**
`assertEquals(P, Q);`	**OK**	**F**	**OK**	**OK**
`assertEquals(P, S);`	**F**	**F**	**OK**	**OK**
`assertFalse(R.equals(P));`	**OK**	**OK**	**F**	**OK**
`assertFalse(T.equals(P));`	**OK**	**OK**	**F**	**OK**

b) In einer anderen Testmethode der Klasse wird ein Container für Polygone angelegt:

```
Set<Polygon> p = new HashSet<Polygon>();
```

Dann werden **nacheinander** einige Polygone in diesen Container aufgenommen. Geben Sie nach jeder Operation die Größe `p.size()` des Containers bei Verwendung der angegebenen Version von Polygon an.

	v1	v2	v3	v4
`p.add(P);`	**1**	**1**	**1**	**1**
`p.add(Q);`	**1**	**2**	**1**	**1**
`p.add(R);`	**2**	**3**	**1**	**2**
`p.add(S);`	**3**	**4**	**1**	**2**
`p.add(T);`	**4**	**5**	**1**	**3**
`p.add(P);`	**4**	**5**	**1**	**3**

Aufgabe 2: (20 Punkte)

Es soll eine Methode `laenge` entstehen, welche die geometrische Länge eines `Polygons` liefert.

a) Ergänzen Sie zunächst die Testklasse `PolygonTest` aus **Aufgabe 1** um eine Testmethode `testeLaenge`, welche die Länge der Polygone `P`, `R` und `T` und die Länge der zugehörigen geschlossenen Polygone überprüft.

```
@Test
public void testeLaenge() {
    assertEquals(1., P.laenge(), 0.);
    assertEquals(2., R.laenge(), 0.);
    assertEquals(0., T.laenge(), 0.);
    assertEquals(2., P.geschlossen().laenge(), 0.);
    assertEquals(2., R.geschlossen().laenge(), 0.);
    assertEquals(0., T.geschlossen().laenge(), 0.);
}
```

b) Implementieren Sie die Methode `laenge` für die Klasse `Polygon` in der Version `v2`.

```
public double laenge() {
    double erg = 0.;
    for (int i = 1; i < mPunkte.size(); i++) {
        Punkt a = mPunkte.get(i-1);
        Punkt b = mPunkte.get(i);
        erg += a.abstandZu(b);
    return erg;
}
```

Aufgabe 3: (20 Punkte)

Die Aufgabe bezieht sich auf die Version `v3` von `Polygon` und die Testklasse `PolygonTest` aus **Aufgabe 1**.

a) Diese liefert teilweise ziemlich merkwürdige Ergebnisse. Prüfen Sie die Tests in der linken Spalte **unabhängig** voneinander. Geben Sie wie in Aufgabe 1 als Ergebnis **OK** oder **F** (Failure) an.

`assertFalse(P.istGeschlossen());`	**OK**
`assertTrue(R.istGeschlossen());`	**F**
`assertEquals(R, P.geschlossen());`	**OK**
`assertTrue(P.geschlossen().istGeschlossen());`	**OK**

b) Selbst die Polygon-Punkte verhalten sich höchst eigenwillig. Der folgende Test geht schief:

```
@Test
public void testePunkte() {
    Punkt a = P.anfangsPunkt();
    Punkt b = P.endPunkt();
    double ab = a.abstandZu(b);
    double ba = b.abstandZu(a);
    assertEquals(ab, ba, 1.e-6);
}
```

Und das liegt nicht an einer zu kleinen Toleranz. Woran dann?

Der Anfangspunkt ist vom Typ Polygon.

Bei a.abstandZu(b) wird also nicht die Methode von Punkt

gerufen, sondern die in Polygon ueberschriebene Methode.

Diese liefert den minimalen Abstand der Punkte von a zu b.

Dieser ist, da b der Endpunkt ist, 0.

Aufgabe 4: (20 Punkte)

a) Bei Verwendung der Version `v4` in einer (beliebigen) anderen Klasse kann jedes Polygon in einen illegalen Zustand mit weniger als 2 Polygon-Punkten gebracht werden. Die Initialisierung eines Polygons `p` geschehe mittels

```
Punkt a = Punkt.URSPRUNG;
Punkt b = new Punkt(10., 5.);
Punkt c = new Punkt(-10., 0.);
final Polygon p = new Polygon(a, b, c);
```

Ergänzen Sie diesen Code so, dass `p` am Ende illegal ist.

```
p.remove(0);
p.remove(0);
// oder:
p.clear();
```

b) Auch bei Verwendung von Version `v2` kann ein ordnungsgemäß erzeugtes Polygon nachträglich zerstört werden. Die Initialisierung muss dabei aber anders geschehen. Erzeugen Sie ein Polygon `q` und bringen Sie es dann in einen illegalen Zustand.

```
Punkt a = Punkt.URSPRUNG;
List<Punkt> aa = new ArrayList<Punkt>();
for (int i = 0; i < 2; i++) aa.add(a);
Polygon q = new Polygon(aa);
aa.remove(0);
```