Java Arbeitsblatt 2: Arrays, Klassen, Objekte

1 Arrays

1. Erstelle im Ordner *Projects* einen Unterordner *Arbeitsblatt_2*
2. Öffne Visual Studio Code und öffne (Ctrl&K, Ctrl&O) den Unterordner *Arbeitsblatt_2*
3. Erstelle eine Datei mit dem Namen *SerialHello.java* und fülle sie mit dem folgenden Code:
```
public class SerialHello {
    public static void main(String[] args) {
        String[] names = {"Anna", "Bello", "Cara", "Delta"};
        for (int i = 0; i < names.length; i++) {
            System.out.println("Hello " + names[i] + "!");
        }
    }
}
```
Teste das Programm.
4. `String[] names` definiert ein Array von Strings, d.h. eine numerierte Menge von Strings. Seine Elemente werden mit `names[0]` bis `names[3]` angesteuert. Es hat eine Eigenschaft `length` vom Typ `int`, das angibt, wie viele Elemente es enthält.
5. Mache Experimente, um zu verstehen, wie das Array funktioniert. Füge z.B. noch mehr Namen hinzu.
6. Auch das Argument `args` der Main-Methode ist ein `String`-Array. Benutze diese Information, um das Programm so umzuschreiben, dass der Kommandozeilen-Aufruf `SerialHello Alice Bob Mallory` die Ausgabe
```
Hello Alice!
Hello Bob!
Hello Mallory!
```
erzeugt.
7. Arrays können auch von anderen Datentypen erstellt werden, z.B. `int` oder `double` oder von nicht-primitiven Datentypen. Die Syntax ist (am Beispiel eines `int`-Arrays):

`int[] a;`	Deklaration des Arrays `a` als `int`-Array
`a = new int[3];`	Erzeugung eines `int`-Arrays der Grösse 3 (das Array wird mit Inhalt `{0, 0, 0}` initialisiert). Zuweisung des Arrays zur Variablen `a`.

`a[0] = 5;`
`a[1] = 2 + 2;`
`a[2] = a.length;`
	Füllen des Arrays, so dass es danach den Inhalt `{5, 4, 3}` hat.

`a = new int[2] {1, 1};`	Erzeugung eines neuen `int`-Arrays der Grösse 2, initialisiert mit Inhalt `{1, 1}`. Zuweisung des Arrays zur Variablen `a`.

`int[] a = {1, 2, 1};`	Deklaration, Erzeugung und Füllen des Arrays in einem Schritt (**Achtung:** funktioniert nur, wenn `a` noch *nicht* deklariert wurde!)
~~`a = {1, 2, 1};`~~	Funktioniert nicht, auch wenn das Array schon deklariert wurde.

8. „Zweidimensionale Arrays“ entstehen, indem wir Arrays von Arrays erstellen. So können zweidimensionale Situationen (Matrizen, Spielfelder für Spiele wie Schach, Tetris, Schiffe versenken etc.) einfach festgehalten werden.
   Erstelle eine Klassendatei TicTacToe.java mit der main-Klasse mit folgendem Code:
   ```
   public static void main(String[] args) {
       char[][] playfield = {{'X', 'X', 'O'},
           {'O', 'O', 'X'},
           {'X', 'O', 'O'}};
       for (int y = 0; y < 3; y++) { //Zeile für Zeile
           for (int x = 0; x < 3; x++) {
               System.out.print(playfield[x][y]);
           }
           System.out.println("");
       }
   }
   ```
   Führe das Programm aus und stelle sicher, dass du alles verstanden hast.
   Bitte beachte die Orientierung des Koordinatensystems (Null links oben, y-Achse geht nach unten)!
   Beachte auch, dass das Array ein Array von Spalten-Arrays ist, was (auf den ersten Blick unerwartet) eine „gespiegelte“ Ausgabe ergibt.

9. Lies den folgenden Abschnitt aus dem Buch „Sprechen Sie Java?“ von Hanspeter Mössenböck. Mache ggf. Experimente, um das beschriebene Verhalten auszuprobieren.

Arrayzuweisung

Einer Arrayvariablen dürfen alle Arrays zugewiesen werden, die vom passenden Elementtyp sind. Dem Array

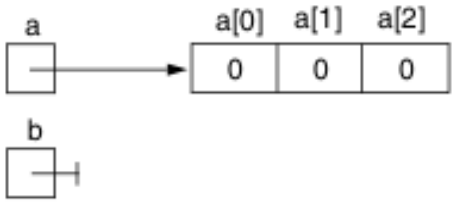
```
int[] a;
```

dürfen also beliebige int-Arrays zugewiesen werden, aber keine float-Arrays. Bei der Zuweisung wird jedoch nicht der *Wert* des Arrays in a gespeichert, sondern nur seine *Adresse*. Mehrere Arrayvariablen können daher auf dasselbe Array zeigen. Das ist für Programmieranfänger oft schwer zu verstehen, weshalb wir uns die Erzeugung und Zuweisung von Arrays nochmals anhand eines Beispiels anschauen. Im Codestück

```
int[] a, b;
a = new int[3];
```

werden zwei Arrayvariablen a und b deklariert, aber nur a zeigt auf ein Array, b ist noch leer. Die Variable b enthält den Wert null. null ist ein vordefinierter Zeigerwert, der so viel bedeutet wie »zeigt nirgendwo hin«. Er darf nicht mit dem int-Wert 0 verwechselt werden.

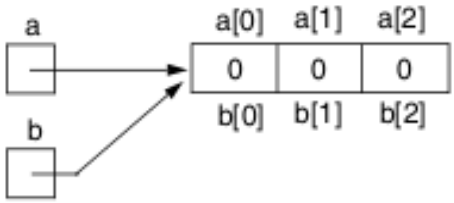
Elemente eines neu erzeugten Arrays werden in Java automatisch initialisiert. Numerische Elemente bekommen den Wert 0, boolean-Elemente den Wert false. Damit ergibt sich folgendes Bild (das Zeichen ⊣ bedeutet den Wert null):



Wir können nun die Zuweisung

```
b = a;
```

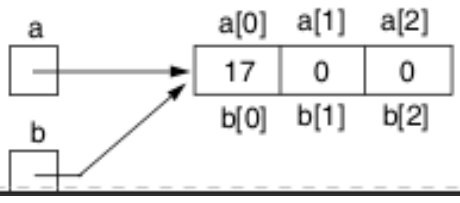
durchführen, was erlaubt ist, weil a auf ein int-Array zeigt und b ebenfalls int-Arrays referenzieren kann. Damit ergibt sich folgendes Bild:



a und b zeigen jetzt auf das gleiche Array. Die Zuweisung ist also eine Zeigerzuweisung. Nur die in a gespeicherte Adresse des Arrays wird b zugewiesen, nicht das Array selbst. Arrayzuweisungen sind in Java immer Zeigerzuweisungen!

Die Elemente des Arrays können jetzt nicht nur als a[0] oder a[2] angesprochen werden, sondern auch als b[0] oder b[2]. Weist man a[0] einen neuen Wert zu, so ändert sich auch der Wert von b[0]. Nach der Zuweisung

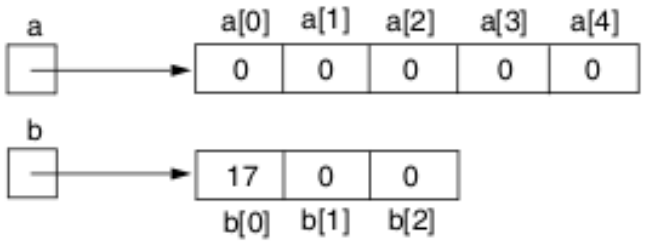
a[0] = 17; sieht das Array so aus:



Wenn man a nun ein neues Array zuweist, z.B.

```
a = new int[5];
```

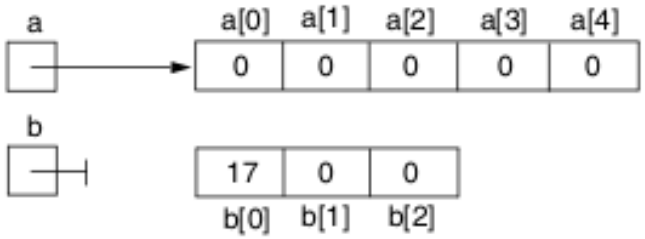
so ergibt sich folgendes Bild:



a und b zeigen jetzt wieder auf verschiedene Arrays. Interessant ist auch die Zuweisung

```
b = null;
```

Der Wert null kann jeder Arrayvariablen zugewiesen werden, wenn man will, dass die Variable auf kein Array zeigt. Damit ergibt sich folgendes Bild:



Das Array, auf das b zeigte, hängt nun »in der Luft«. Es wird von keiner Arrayvariablen mehr referenziert, und man kann es nicht mehr ansprechen. Mehr noch: Es gibt in diesem Beispiel keine Möglichkeit, je wieder einen Zeiger auf dieses Array verweisen zu lassen. Also wird es nicht mehr gebraucht und kann weggeworfen werden. Wenn ein Array nicht mehr referenziert wird, sorgt das Java-System automatisch dafür, dass sein Speicherplatz wieder freigegeben wird und für weitere Speicheranforderungen zur Verfügung steht.

P.S. Diesen Vorgang nennt man „Garbage Collection“, ein Prozess, der in Java (im Vergleich zu anderen Programmiersprachen) automatisch passiert (übrigens auch, wenn ein Objekt/ eine Variable am Ende einer Prozedur/ Schleife etc. das Ende seiner Lebensdauer erreicht hat).

Übrigens: Das Verhalten, dass mit = und == auf den *Pointer* (Zeiger) Bezug genommen wird statt auf den Inhalt, zeigen auch Objekte (s. nächster Abschnitt). Deshalb gibt es dafür die `equals()`-Methoden.

2 Klassen und Objekte

1. *Objekte* werden mit dem Schlüsselwort `new` erzeugt. Dabei müssen wir angeben, welcher Art (*Klasse*) das Objekt ist, z.B. `new String` oder `new java.awt.Point` (vordefinierte Klasse aus der Java-Objektbibliothek). Die Klasse bestimmt, was das Objekt für Eigenschaften und Methoden hat. Ein String-Objekt hat zum Beispiel die Methode `lenght()`, die die seine Länge zurückgibt. Für Klassen gibt es sogenannte UML-Diagramme, die beschreiben, welche Eigenschaften und Methoden sie aufweisen. In Abb. 1 siehst du ein UML-Diagramm für `java.awt.Point`. Solch ein Punkt hat:

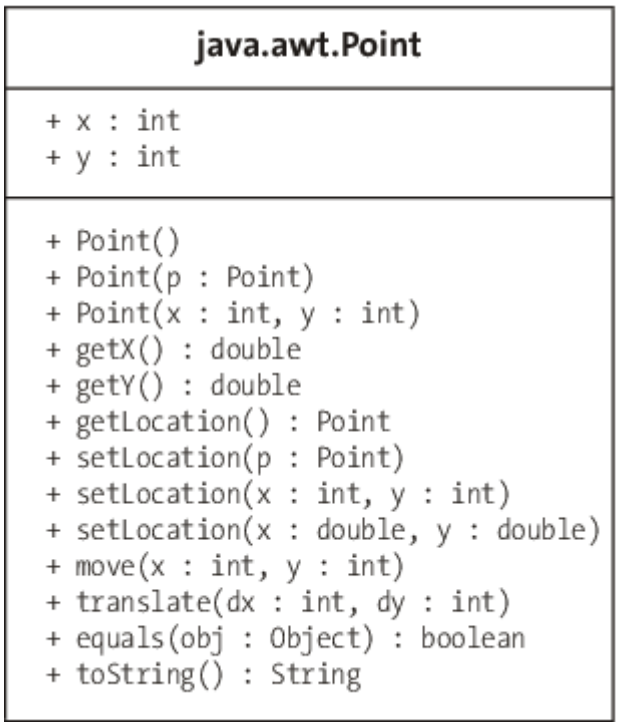


Abb. 1: UML Class Diagram für java.awt.Point

   - die *Eigenschaften* (*Felder*) `x` und `y` (die seine Position angeben – Achtung, nur Ganzzahlen (`int`)).
   - Methoden:
     - Konstruktoren:
       `Point p = new Point();` erzeugt einen neuen Punkt mit Koordinaten (0,0).
       `Point q = new Point(p);` erzeugt eine Kopie von Punkt p.
       `Point r = new Point(2, 5);` erzeugt einen neuen Punkt mit Koordinaten (2,5).
     - getter- und setter-Methoden:
       `p.setLocation(3, 4);` kann anstelle von `p.x = 3; p.y = 4;` verwendet werden, während `p.getX()` die x-Koordinate von `p` zurückgibt (seltsamerweise als `double`, was im Moment noch nicht logisch sein muss).
       Der tiefere Sinn davon ist, dass in den meisten Klassen (`java.awt.Point` ist hier eine Ausnahme) die Felder nicht direkt zugänglich sind und deshalb nicht direkt gelesen oder verändert werden können. Das erlaubt, unerwartetes Verhalten zu vermeiden, indem die setter-Methoden Kontrollen ausführen, bevor sie den Feldern gewisse Werte zuweisen.
     - Die `move`-Methode (identisch zu `setLocation`) und die `translate`-Methode (Verschiebung um `dx`, `dy`)
     - `equals`-Methode, die feststellt, ob zwei Punkte gleich sind (die gleiche Position haben, *nicht* den gleichen Zeiger!) und `toString`-Methode (jedes Objekt hat eine solche, auch wenn sie nicht immer die erwarteten Dinge tut)

2. Aus „Java ist auch eine Insel“ von Christian Ullenboom:

Ein Klassiker aus dem Genre der Computerspiele ist *Snake*. Auf dem Bildschirm gibt es den Spieler, eine Schlange, Gold und eine Tür. Die Tür und das Gold sind fest, den Spieler können wir bewegen, und die Schlange bewegt sich selbstständig auf den Spieler zu. Wir müssen versuchen, die Spielfigur zum Gold zu bewegen und dann zur Tür. Wenn die Schlange uns vorher erwischt, haben wir Pech gehabt, und das Spiel ist verloren.

Vielleicht hört sich das auf den ersten Blick komplex an, aber wir haben alle Bausteine zusammen, um dieses Spiel zu programmieren:

- Spieler, Schlange, Gold und Tür sind `Point`-Objekte, die mit Koordinaten vorkonfiguriert sind.
- Eine Schleife läuft alle Koordinaten ab. Ist ein Spieler, die Tür, die Schlange oder Gold »getroffen«, gibt es eine symbolische Darstellung der Figuren.
- Wir testen drei Bedingungen für den Spielstatus: 1. Hat der Spieler das Gold eingesammelt und steht auf der Tür? (Das Spiel ist zu Ende.) 2. Beißt die Schlange den Spieler? (Das Spiel ist verloren.) 3. Sammelt der Spieler Gold ein?
- Mit dem `Scanner` können wir auf Tastendrücke reagieren und den Spieler auf dem Spielbrett bewegen.
- Die Schlange muss sich in Richtung des Spielers bewegen. Während der Spieler sich nur entweder horizontal oder vertikal bewegen kann, erlauben wir der Schlange, sich diagonal zu bewegen.

Die `Point`-Eigenschaften, die wir nutzen, sind:

- Objektzustände `x`, `y`: Der Spieler und die Schlange werden bewegt, und die Koordinaten müssen neu gesetzt werden.
- Methode `setLocation(...)`: Ist das Gold aufgesammelt, setzen wir die Koordinaten so, dass die Koordinate vom Gold nicht mehr auf unserem Raster liegt.
- Methode `equals(...)`: Testet, ob ein Punkt auf einem anderen Punkt steht.

Öffne die Datei ZZZZZnake.java und führe sie aus. Versuche aus dem Quellcode zu verstehen, wie das Spiel funktioniert. Beschreibe danach, was in den mit `// 1)` bis `// 6)` markierten Zeilen passiert.[i]

3. Falls du das Programm weiterentwickeln willst, hier ein paar Ideen (nach Ullenboom):
   - Spieler, Schlange, Gold und Tür sollen auf Zufallskoordinaten gesetzt werden (mit `(int) (Math.random() * 40)`. Dabei erzeugt `Math.random()` eine `double`-Zufallszahl zwischen 0 und 1, und `(int)` konvertiert die Zahl zu `int` (durch Abrunden).
   - Statt nur eines Stücks Gold soll es zwei Stücke geben.
   - Statt einer Schlange soll es zwei Schlangen geben.
   - Mit zwei Schlangen und zwei Stücken Gold kann es etwas eng für den Spieler werden. Er soll daher am Anfang 5 Züge machen können, ohne dass die Schlangen sich bewegen.
   - Für Vorarbeiter: Das Programm, das bisher nur eine Methode ist, soll in verschiedene Untermethoden aufgespalten werden.

3 Klassen selber erstellen – Setter und Getter, Konstruktoren, `toString()` und `equals()`

1. Wir wollen eine Klasse `Fraction` schreiben, mit der wir rechnen können. Dazu erstellen wir eine Datei *Fraction.class* mit dem Inhalt
   ```
   public class Fraction {
       int numerator, denominator;
   }
   ```
   Ebenfalls erstellen wir eine Datei *FractionTester.java* (Achtung, muss im gleichen Ordner sein!) mit einer `main`-Methode:
   ```
       Fraction f = new Fraction();
       f.numerator = 12;
       f.denominator = 0;
       System.out.println(f.numerator + "/" + f.denominator);
   ```
   Die Klasse `Fraction` ist sozusagen die Vorlage, nach der ein konkretes `Fraction`-Objekt `f` erzeugt wird. Wir nennen `f` auch eine *Instanz* der Klasse `Fraction`. Beim Compilieren und Ausführen funktioniert alles ohne Probleme. Allerdings macht das mathematisch keinen Sinn.

2. Um solchen Quatsch abfangen zu können, soll der Wert `denominator` nicht von aussen veränderbar sein, sondern nur von Prozeduren innerhalb der Klasse. Wir ändern deshalb die Zeile in *Fraction.java* zu
   ```
   private int numerator, denominator;
   ```
   Jetzt können wir zwar ein Fraction-Objekt `f` erstellen, aber aus `FractionTester` keine Werte mehr zuweisen oder auslesen (Ausprobieren, Fehlermeldung lesen!). Ziemlich doof, oder?

3. Ergänze die Klasse Fraction mit den folgenden Methoden:
   ```
   public int getNumerator() {
       return numerator;
   }

   public int getDenominator() {
       return denominator;
   }
   ```

```
public void setNumerator(int numerator) {
    this.numerator = numerator;
}

public void setDenominator(int denominator) {
    if (denominator == 0) {
        throw new ArithmeticException("Division by zero!!");
    } else {
        this.denominator = denominator;
    }
}
```

Hier meint `this` das `Fraction`-Objekt, dem wir den Wert zuweisen. Es wird geschrieben zur Unterscheidung des *Feldes* `numerator` des `Fraction`-Objekts (`this.numerator`) vom *Argument* `numerator` der Methode `setNumerator()`.
Der Code in der `FractionTester`-Klasse muss dann entsprechend angepasst werden:

```
Fraction f = new Fraction();
f.setNumerator(12);
f.setDenominator(0);
System.out.println(f.getNumerator() + "/" + f.getDenominator());
```

Bitte ausprobieren! Das Programm „wirft“ also unsere selbst generierte Fehlermeldung. Mit `f.setDenominator(15);` funktioniert aber alles wie erwartet.

Getter- und Setter-Methoden sind der übliche Weg, um mit Feldern (Variablen) von Objekten direkt zu interagieren. Deshalb bieten IDEs die Möglichkeit, diese Methoden automatisiert zu erstellen. (In VSC Rechtsklick auf den Code – *Source Action/ Generate Getters and Setters*)

4. Eine Standardmethode von Objekten ist `toString()`. Ersetze den `print`-Code in `FractionTester`-Klasse durch
   `System.out.println(f.toString());`
   Beim Ausführen kommt nichts wirklich Brauchbares heraus. Die `toString()`-Methode eines Objektes ist standardmässig von der Form „Klasse@Hashcode“[ii]. Wir wollen aber den „Wert“ des Bruchs als Zähler und Nenner zurückgeben. Also schreiben wir in der Klasse `Fraction` eine Methode
   ```
   @Override
   public String toString() {
       return "[" + numerator + "/" + denominator + "]";
   }
   ```
   Nun kommt etwas Sinnvolleres raus.
   `@Override` heisst übrigens, dass die `toString()`-Methode der Klasse `Object` überschrieben wird.

5. Das Erzeugen von Brüchen mit einzelner Zuweisung der Werte von `numerator` und `denominator` ist etwas mühsam. Wenn wir eine sogenannte *Konstruktor-Methode* schreiben, können wir den Bruch erzeugen und initialisieren in einem Schritt. Eigentlich sollten Konstruktor-Methoden Initialisierungs-Methoden heissen, denn das Objekt wurde bereits vorher erzeugt. Konstruktor-Methoden werden normalerweise am Anfang der Klasse positioniert, gleich nach der Definition der Felder. Wir schreiben also:
   ```
   public Fraction(int numerator, int denominator) {
       this.numerator = numerator;
       this.setDenominator(denominator);
   }
   ```
   Hier sehen wir wieder die Zuordnung mit `this`. In der zweiten Zeile benutzen wir die

Methode `setDenominator()`, da diese bereits die Überprüfung auf Nullwerte enthält. Somit müssen wir diesen Code nicht nochmals schreiben.
Wenn du jetzt `FractionTester` ausführst, bekommst du einen Fehler. Dadurch, dass wir einen neuen Konstruktor geschrieben haben, wird der Standard-Konstruktor `Fraction()` ohne Argument ungültig. Das ist aber eine gute Sache. Mit `new Fraction()` wurde nämlich ein `Fraction`-Objekt mit nicht-initialisierten Feldern erzeugt. Nicht-initialisierte `int`-Variablen haben standardmässig den Wert 0, d.h. unser Bruch `f` hatte den Wert 0/0, bevor wir ihm Werte zugewiesen haben…
Also schreiben wir neu:
`Fraction f = new Fraction(12, 15);`
und haben erst noch zwei Zeilen Code gespart. Teste, ob die so erzeugte `Fraction` korrekt auf der Konsole ausgegeben wird.

6. Um das Kopieren von Brüchen zu vereinfachen (da ja die Zuweisung `g = f` in
```
Fraction f = new Fraction(1, 2);
Fraction g = f;
f.setNumerator(5);
System.out.println(g); // [5/2]
```
eine Zeigerzuweisung ist und im Speicher nur ein `Fraction`-Objekt existiert, auf das beide Variablen verweisen), erstellen wir einen *Copy-Konstuktor*:
```
public Fraction(Fraction f) {
    this(f.numerator, f.denominator));
}
```
Dieser erstellt ein neues `Fraction`-Objekt und kopiert die Werte der übergebenen `Fraction f`. Die Methode `this()` steht für den Aufruf der Konstruktormethode `Fraction(n, d)`. Dieser Aufruf muss in Konstruktoren immer an erster Stelle stehen.
Die zweite Zeile im obigen Test-Code kann nun durch
```
Fraction g = new Fraction(f);
```
ersetzt werden, und die Änderung von `f.numerator` beeinflusst `g` nicht mehr.

7. Wenn wir wollen, können wir auch einen argumentlosen Konstruktor für einen „Standard-Bruch“ schreiben:
```
public Fraction() {
    this(0, 1);
}
```

8. Ebenfalls Standard ist die `equals()`-Methode, die feststellt, ob zwei Objekte gleich sind. Diese Methode wird gebraucht, da `f == g` nur die Zeiger vergleicht und nicht den Inhalt (s.o. bei den Arrays). Wir wollen, dass die Brüche 12/15 und -3/-4 als gleich angesehen werden. Dies erreichen wir mit:
```
public boolean equals(Fraction f) {
    return this.numerator * f.denominator == this.denominator * f.numerator;
}
```
Teste sie in der FractionTester-Klasse, indem du ein Fraction-Objekt g erzeugst und mit -3/-4 initialisierst und danach
`System.out.println(f.equals(g));`
ausführst.

9. Jetzt fehlen noch Operationen wie Addition, Subtraktion etc. Wir implementieren
```
public void add(Fraction f) {
    this.setNumerator(this.numerator * f.denominator + this.denominator * f.numerator);
    this.setDenominator(this.denominator * f.denominator);
}
```
Aufgerufen werden kann die Methode durch
`f.add(g);`
Bitte beachten: der Rückgabetyp ist `void`, denn das Resultat wird unter `f` gespeichert.
Bitte mit geeignetem Code testen!

10. Alle unsere Methoden in der Klasse sind bisher Instanzen-Methoden, d.h. an eine Instanz `f` gebunden (deshalb auch der Aufruf über die Syntax `f.methode()`). Im Gegensatz dazu sind *statische* Methoden von konkreten Objekten unabhängig aufrufbar, da sie sich auf die ganze Klasse beziehen. Um festzulegen, dass eine Methode statisch ist, brauchen wir das Schlüsselwort `static`:
```
public static Fraction add(Fraction f, Fraction g) {
    f.add(g);
    return f;
}
```
In der `FractionTester`-Klasse rufen wir die Methode durch `Fraction.add(f, g);` auf. Achtung: Der Rückgabetyp ist dieses Mal `Fraction`, nicht `void`. Bitte testen!

11. Es gibt auch die Möglicheit, statische Variablen zu definieren. Wir können zum Beispiel `static int numberOfFractions` definieren, die zählt, wie viele Fraction-Objekte erzeugt wurden. Im Konstruktor muss dann `numberOfFractions++;` stehen. Probiere es aus!

12. Noch ein Schlüsselwort: `final` bezeichnet Konstanten, also Variablen die sich nicht ändern. Definiere in der Testklasse eine Konstante mit `final double PI = 3.14159;` und versuche ihren Wert zu ändern. Sind Methoden `final`, so können sie in abgeleiteten Klassen nicht überschrieben werden.

13. Etwas zu **Namenskonventionen**:
    - *Schlüsselwörter* werden klein geschrieben: `public, static, for, if`
    - *primitive Datentypen* ebenso: `int, float, char`
    - *Klassennamen* sind Substantive (da sie Vorlagen für Objekte sind) und beginnen mit einem Grossbuchstaben: `Fraction, String, Tester, FractionTester`
    - *Methoden* tun etwas und haben deshalb Verbnamen (Camel Case: klein beginnen, danach jedes Wort gross): `getN(), add(), move(), killAllOpenWindows()`
    - *Variablennamen* verwenden ebenfalls Camel Case: `variable, myVariable, numberOfElements, this`
    - *Konstanten* werden durchgehend gross geschrieben: `EULERS_NUMBER, EARTH_CIRCUMFERENCE`

14. Die **Sichtbarkeit** von Variablen und Methoden kann mit den Schlüsselwörtern `public`, `protected` und `private` beeinflusst werden (s. Tabelle).

Access Modifiers

Modifier	Class	Package	Subclass	Global
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

i Lösung:
1) Ein neues `java.awt.Point-Objekt` wird erstellt und der Variablen `playerPosition` zugeordnet. Dabei wird der Konstruktor `Point(int x, int y)` aufgerufen, das die Koordinaten setzt.
2) `while (true) {...}` beschreibt eine Endlosschleife (alternative Möglichkeit: `for (;;) {...}`). Die einzige Möglichkeit, die Schleife zu beenden, ist mit einem `return`-Befehl (s. 4))
3) Es wird ein neuer `Point p` erstellt. Da seine Lebensdauer nur bis zum Ende der inneren for-Schleife ist, kann (und muss) er bei jedem neuen Durchlauf der Schleife neu erstellt werden.
4) siehe 2)
5) Die `max`- und `min`-Anweisungen stellen sicher, dass das Spielfeld nicht verlassen wird…
6) `x--` heisst „`x` um eins vermindern“ und ist das Gegenteil des `++`-Operators.

ii Hashcode = effizient, aber nicht rückverfolgbar berechneter Ausdruck, der möglichst einzigartig (und damit eindeutig) für das betreffende Objekt ist.