5 B2a: Objektorientierte Programmierung

5.1 Klassen und Objekte

Objekte (*objects*) werden mit dem Schlüsselwort `new` erzeugt. Dabei müssen wir angeben, welcher Art (**Klasse (*class*), Typ (*type*)**) das Objekt ist, z.B. `new String` oder `new java.awt.Point` (vordefinierte Klasse aus der Java-Klassenbibliothek). Wir sagen auch, das Objekt sei eine **Instanz (*instance*)** der Klasse `String`/ `java.awt.Point`.

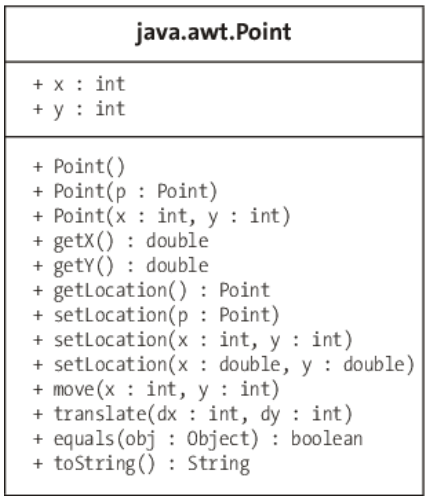

Abbildung 1: UML Class Diagram für `java.awt.Point`

Die Klasse bestimmt, was das Objekt für Eigenschaften und Methoden hat (ein `String`-Objekt hat zum Beispiel die Methode `length()`, die seine Länge zurückgibt). Diese werden in UML (*Unified Markup Language*)-Diagrammen dargestellt. In Abbildung 1 siehst du das UML-Diagramm für `java.awt.Point`. Die Klasse hat:

- Die **Eigenschaften/ Attribute (*attributes*)/ Felder (*fields*)** `x` und `y`, die für seine Position im Koordinatensystem stehen (komischerweise nur Ganzzahlen).
- **Konstruktor**-Methoden (**Konstruktoren/ *constructors***):

`Point p = new Point(2, 5);`	erzeugt einen neuen Punkt mit Koordinaten $(2, 5)$.
`Point q = new Point();`	erzeugt einen neuen Punkt mit Koordinaten $(0, 0)$.
`Point r = new Point(p);`	erzeugt eine Kopie eines Punktes `p` (d.h. im Speicher wird tatsächlich ein zweites Objekt angelegt und nicht nur ein neuer Zeiger auf das gleiche Objekt).

- **Objektmethoden (*instance methods*)**, d.h. Methoden, die mit der Syntax `p.methode()` auf einem Objekt `p` vom Typ `Point` aufgerufen werden:
  - **Getter- und Setter-Methoden**:

    `p.setLocation(3, 4);` macht das Gleiche wie `p.x = 3; p.y = 4;`

    `p.getX()` gibt die x-Koordinate von `p` zurück (seltsamerweise als `double` statt als `int`).

    Getter- und Setter-Methoden existieren, da in den meisten Klassen (`java.awt.Point` ist hier eine Ausnahme) die Felder auf `private` gesetzt sind und deshalb von ausserhalb der Klasse nicht direkt gelesen oder verändert werden können. Dieses **Data Hiding** ist Teil des Prinzips der **Encapsulation**: Sie erlaubt, unerwartetes Verhalten zu vermeiden, indem z.B. Setter-Methoden Kontrollen ausführen, bevor sie den Feldern gewisse Werte zuweisen (s. Abschnitt 5.2). Der andere Vorteil der Encapsulation ist das Verbergen der inneren Struktur/ Mechanismen vor den Anwendern der Klasse, was erlaubt, die Implementation im Innern der Klasse zu ändern, solange das äussere Verhalten gleich bleibt.[6]
  - Eine **`toString()`-Methode**, die das Objekt in einer sinnvoll lesbaren Form in einen `String` verwandelt (statt einen Hashcode/ Identifier[7] auszugeben). Das sieht dann je nach Java-Implementation etwa so aus: `java.awt.Point[x=10,y=9]`.
  - Eine `equals()`-Methode, die feststellt, ob der Punkt gleich einem anderen ist. Dabei wird nicht wie mit `==` Objektgleichheit geprüft, sondern Inhaltsgleichheit, d.h. ob die beiden Punkte die gleichen Feldinhalte (also die gleichen Koordinaten) haben.
  - Klassenspezifische Methoden:
    - `move()` und `translate()`, die das Erwartbare tun.
    - `p.getLocation()` macht das Gleiche wie `new Point(p)`.

      `q.setLocation(p);` macht das Gleiche wie `q.x = p.x; q.y = p.y;` (also eine Übernahme der Koordinaten, nicht der Zeiger).

      Die Methoden existieren hauptsächlich aus Kompatibilitätsgrün-

[6] Zu Encapsulation und den anderen drei Prinzipien der objektorientierten Programmierung siehe https://stackify.com/oops-concepts-in-java/

[7] Hashcode = effizient, aber nicht rückverfolgbar berechneter Ausdruck, der möglichst einzigartig (und damit eindeutig) für das betreffende Objekt ist. Hashcodes werden gebraucht, um Objekte quasi-eindeutig zu identifizieren, deshalb der Begriff *identifier*.

den: Bei den meisten anderen geometrischen Objekten (wie z.B. `java.awt.Rectangle`) sind das Objekt und die „Location“ zwei unterschiedliche Dinge.

- Die Klasse hat keine **statischen/ Klassenmethoden (*static/ class methods*)**, d.h. Methoden, die unabhängig von einem konkreten `Point`-Objekt sind und deshalb mit der Syntax `Klasse.methode()` aufgerufen werden. Beispiele dafür aus anderen Klassen wären z.B. `Math.random()` (da muss nicht zuerst ein `Math`-Objekt erstellt werden), `Arrays.toString()` etc. Solche wären im UML-Diagramm unterstrichen dargestellt.

Aufgabe 1. Schreibe eine Klasse `PointTester.java`. Importiere die Klasse `java.awt.Point` vor dem Klassencode. Dann:

1. Erzeuge ein `Point`-Objekt mit Koordinaten $(2, 5)$, speichere es in einer Variablen und gib seinen Inhalt auf der Konsole aus (indem du implizit oder explizit `toString()` aufrufst).
2. Vertausche die beiden Koordinaten des Punktes. Greife dabei nicht direkt auf seine Felder zu, sondern brauche die Getter- und Setter-Methoden.
3. Erzeuge mit möglichst wenig Code einen zweiten Punkt, der die Koordinaten des ersten übernimmt. Teste die beiden Punkte auf Objekt- und Inhaltsgleichheit.

5.2 `Fraction` - Eigene Klassen schreiben, Encapsulation

5.2.1 Grundlagen

Wir wollen eine Klasse `Fraction` schreiben, mit der wir rechnen können. Eine Klasse `Fraction.java` mit den Feldern `numerator` und `denominator` sowie eine Datei `FractionTester.java` für Testcode existieren bereits.

Probiere den Testcode in `FractionTester` aus. Es wird ein konkretes `Fraction`-Objekt `f` nach der „Vorlage“ der Klasse `Fraction` erstellt. Stelle sicher, dass du alles verstehst.

toString() Die Zeile mit dem `println()`-Befehl ist etwas umständlich, aber leider liefert `System.out.println(f)` ja nur einen Hashcode. Das wollen wir ändern!

Füge der Klasse `Fraction` eine Funktion `toString()` hinzu. Am Einfachsten geht das mit *Rechtsklick/ Source Action/ Generate toString()*. Passe den Code der automatisch generierten `toString()`-Methode an, um eine sinnvolle Bruchausgabe zu bekommen.

`@Override` heisst, dass die `toString()`-Methode der Klasse `Object` (die einfach nur den Hashcode generiert und ausgibt) überschrieben wird.

5.2.2 Encapsulation

Encapsulation, Getter und Setter: Zeile 6 in `FractionTester` macht mathematisch natürlich keinen Sinn. Genau um solchen Quatsch abfangen zu können, soll der Wert `denominator` nicht von aussen veränderbar sein, sondern nur von Prozeduren innerhalb der Klasse (die aber ihrerseits `public` sind, also von aussen aufgerufen werden können). Setze deshalb das Schlüsselwort `private` vor `int numerator, denominator;`. Nun funktionieren die Zeilen 5-7 im Testcode nicht mehr. Probiere es aus und studiere die Fehlermeldung!

Wir brauchen nun also Methoden, um auf `numerator` und `denominator` zuzugreifen. Eine davon, `setDenominator()`, existiert bereits. Sie produziert einen Fehler, wenn der Nenner auf 0 gesetzt wird. Probiere es in der Tester-Klasse aus! Studiere danach den Code. Das Schlüsselwort `this` ist ein Platzhalter für das Objekt, auf dem die Methode aufgerufen wird. Dadurch kann das Methodenargument `denominator` vom Feld `this.denominator` des `Fraction`-Objekts unterschieden werden.

Erstelle nun die fehlenden Getter- und Setter-Methoden. Am Einfachsten geht das per *Rechtsklick/ Source Action/ Generate Getters* bzw. *Setters*. Studiere den generierten Code kurz. Weise danach dem `Fraction`-Objekt `f` im Testcode gültige Werte für Zähler und Nenner zu.

Hintergrundwissen: Die **Sichtbarkeit** von Variablen und Methoden kann mit den Schlüsselwörtern (***access modifiers***) `public` (+), `protected` (#) und `private` (-) beeinflusst werden:

sichtbar…	+	#	~	–
…in der Klasse selbst	ja	ja	ja	ja
…in Klassen des gleichen Pakets	ja	ja	ja	-
…in abgeleiteten Klassen	ja	ja	-	-
…global	ja	-	-	-

Bemerkungen:

- +, #, ∼ (für fehlenden Modifier) und - sind die Notationen, die in UML-Diagrammen gebraucht werden (s. Abb. 1).
- **Pakete** fassen mehrere Klassen zusammen, die inhaltlich oder funktionell zusammengehören. So ist z.B. `java.awt` ein Paket, das Klassen für User Interfaces und graphische Darstellungen bereitstellt (z.B. `Point.java`).
- **Abgeleitete Klassen (*subclasses*)** sind Erweiterungen von Klassen, die alle Eigenschaften und Methoden der ursprünglichen Klasse erben (sie können aber überschrieben werden). So ist etwa jede Klasse (implizit) von `Object` abgeleitet und erbt Methoden wie z.B. `toString()` oder den argumentlosen Konstruktor. Wenn wir etwa eine Klasse `Square` schreiben würden, könnte sie von einer Klasse `Rectangle` abgeleitet sein, die wiederum von einer Klasse `Quadrilateral` abgeleitet ist.

5.2.3 Konstruktoren und Gleichheit

Konstruktoren: Das Erzeugen von Brüchen mit einzelner Zuweisung der Werte von `numerator` und `denominator` ist mühsam. Wenn wir eine sogenannte Konstruktor-Methode schreiben, können wir den Bruch in einem Schritt erzeugen und initialisieren. Eigentlich sollten Konstruktor-Methoden Initialisierungs-Methoden heissen, denn sie initialisieren lediglich die Felder des soeben erzeugten Objekts. Konstruktor-Methoden werden normalerweise am Anfang der Klasse positioniert, gleich nach der Definition der Felder. Erzeuge einen Konstruktor mit *Rechtsklick/ Generate Constructors*, hake beide Felder an und drücke OK. Studiere den Code.

Hier sehen wir wieder die Zuordnung mit `this`. Ersetze `this.denominator;` durch `this.setDenominator(denominator);`. Indem wir den Setter verwenden, haben wir gleich die Wertprüfung für ungültige Nenner wieder dabei.

Wenn du jetzt `FractionTester` ausführst, bekommst du einen Fehler. Dadurch, dass wir einen neuen Konstruktor geschrieben haben, wird der argumentlose Standard-Konstruktor `Fraction()` 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 bekanntlich standardmässig den Wert 0, d.h. unser Bruch `f` hatte den Wert $\frac{0}{0}$, bevor wir ihm Werte zugewiesen haben…

Erzeuge in der Testklasse ein neues `Fraction`-Objekt, das den neuen Konstruktor braucht.

Erzeuge ausserdem einen argumentlosen Konstruktor `Fraction()` mit einer einzigen Zeile `this(0, 1);`. Die Methode `this(int, int)` steht für den Aufruf der bereits erstellten Konstruktormethode `Fraction(int, int)`. Dieser Aufruf muss in Konstruktoren immer an erster Stelle stehen. Die Initialisierung von `Fraction`-Objekten mit $\frac{0}{1}$ macht mehr Sinn als mit $\frac{0}{0}$. Teste wieder.

Copy-Konstruktor und `equals()`: Die Zuweisung `g = f;` ist zwischen zwei `Fraction`-Variablen bekanntlich lediglich eine Zeigerzuweisung, und im Speicher existiert nur ein `Fraction`-Objekt. Um `Fraction`-Objekte echt zu kopieren, brauchen wir einen Copy-Konstruktor. Entkommentiere dazu den Code `public Fraction(Fraction f){ ... }`.

Der entkommentierte Konstruktor erstellt ein neues `Fraction`-Objekt und übergibt ihm die Werte der übergebenen Fraction `f`. Probiere es aus, indem du damit in der Testklasse eine Kopie `g` von `f` erstellst. Überprüfe wieder Objekt- und Inhaltsgleichheit.

Inhaltsgleichheit konntest du nur „manuell“ im Debugging-Modus oder durch Vergleich der `toString()`-Outputs überprüfen. Um das zu ändern, können wir eine `equals()`-Methode schreiben. Auch diese kannst du einfach entkommentieren. Studiere sie. Warum macht der Code (mathematisch) Sinn?

Teste die Methode danach an `f` und `g`, aber auch mit einem weiteren Bruch, der den gleichen Wert, aber unterschiedlichen Zähler und Nenner hat wie einer der beiden.

5.2.4 Statische und dynamische Methoden

Statische und dynamische Methoden (Klassen- und Instanzenmethoden): Jetzt fehlen noch Operationen wie Addition, Subtraktion etc. Die Addition ist bereits geschrieben und kann entkommentiert werden. Du siehst, dass zwei Methoden `add()` existieren:

- Die **Klassenmethode (=statische Methode)**, die aufgerufen wird als `Fraction.add(f, g)` und das Resultat als (neues) `Fraction`-Objekt zurückgibt. Das Schlüsselwort `static` macht sie zu einer Klassenmethode.
- Die **Objektmethode (=dynamische Methode)**, die mit `f.add(g)` aufgerufen wird und die `g` zu `f` dazuaddiert (und das Resultat direkt in `f` speichert). Sie hat keine Rückgabe.

Je nach Kontext macht die eine oder andere Methode mehr Sinn. Die Klassenmethode kommt zum Zug, wenn bestehende Objekte nicht verändert werden sollen. Die Objektmethode hingegen kann sinnvoll sein, wenn (Speicher-)Effizienz gefragt ist.

Teste beide Methoden.

Statische Variablen: Es gibt die Möglicheit, neben Methoden auch Variablen als statisch zu definieren. Die Variable `numberOfFractions` (die bereits vordefiniert ist) kann z.B. gebraucht werden, um zu zählen, wie viele `Fraction`-Objekte bereits erzeugt wurden. Im Konstruktor müssen wir dann die Zeile `numberOfFractions++;` hinzufügen (nur im ersten Konstruktor, da die anderen ja darauf zugreifen). Probiere es aus!

Konstanten: Um Namenskonfusion zu vermeiden: `static` heisst nicht "unveränderlich". Das Schlüsselwort dafür ist `final`. Es kennzeichnet Konstanten , also Variablen mit unveränderlichem Wert. Auch Methoden können `final` sein, was heisst, dass sie in abgeleiteten Klassen nicht überschrieben werden können. `final` und `static` können unabhängig voneinander gesetzt werden.

Namenskonventionen :

- **Schlüsselwörter** werden klein geschrieben: `public`, `static`, `for`, `if`
- **primitive Datentypen** ebenso: `int`, `float`, `char`
- **Klassen**namen 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):
  `getNumerator()`, `add()`, `move()`, `killAllOpenWindows()`
- **Variablen**namen sind auch in Camel Case:
  `variable`, `myVariable`, `numberOfElements`
- **Konstanten** werden durchgehend gross geschrieben und verwenden (als einzige) Unterstriche als Worttrenner:
  `PI`, `EULERS_NUMBER`, `EARTH_CIRCUMFERENCE`

Aufgabe 2.

1. Definiere in der Testklasse mit `final double PI = 3.14159;` eine Konstante und versuche ihren Wert zu ändern.

2. Erkläre den gesamten Code `public static void main(String[] args)`.

5.2.5 Weitere Aufgaben

Aufgabe 3. Erstelle ein UML-Diagramm für die Klasse `Fraction`. Stelle dabei Sichtbarkeit dar und ob eine Variable/ Methode statisch ist.

Aufgabe 4. Ergänze die Klasse `Fraction` mit weiteren Funktionen. Oft ist nicht nur das Programmieren selbst, sondern bereits das sorgfältige Planen der Implementation eine wertvolle Übung. Möglichkeiten sind:

- Ergänze Methoden für die fehlenden drei Grundrechenarten. Mach das schlau und vermeide zu viel Codewiederholung.
- Schreibe eine Methode `simplify()` (= der korrekte englische Fachbegriff für „kürzen“!).
- Schreibe eine abgeleitete Klasse `MixedNumber` (mit `public class MixedNumber extends Fraction`). Welche Felder musst du zusätzlich einführen, welche Methoden überschreiben/ ergänzen? Mit `super.` kannst du auf Felder und Methoden der Mutterklasse `Fraction` zugreifen.
- Implementiere Encapsulation für die Variable `numberOfFractions`. Was brauchst du dazu alles?