Algorithmen und Datenstrukturen
Java Crashkurs

Jonas Kriegs und Robin Neubauer

Einleitung

Mit dieser Zusammenstellung von Erklärungen, Aufgaben und Quellen bieten wir Ihnen eine Einführung und Überleitung in die Programmiersprache Java an, um Ihnen den Einstieg in Algorithmen und Datenstrukturen zu erleichtern. Der Kurs stellt eine Ergänzung zu AlgoDat dar und ist rein freiwillig. Sie können ihn komplett durcharbeiten oder sich nur die Themen ansehen, mit denen Sie Schwierigkeiten haben. Inhaltlich ist er in drei Wochen gegliedert, die Themen, an die Sie jede Woche herangeführt werden, korrespondieren mit den Themen der Vorlesung. Am Ende des Kurses sollten Ihnen alle Grundlagen von Java bekannt sein, die Sie benötigen, um mit der Sprache programmieren zu können.

Wir setzen voraus, dass Sie grundlegende Programmierkenntnisse besitzen, und ziehen spezifisch Parallelen zu C und wie die Konzepte, die im Modul “Einführung in die Programmierung” vermittelt wurden, in Java übertragen werden können. Allerdings werden alle Konzepte selbstbezogen erklärt, Sie benötigen also nicht zwingend Erfahrung mit der Sprache C.

Wir erklären bewusst nicht jedes Detail der Sprache und fokussieren uns auf die wesentlichen Merkmale. Um die Ressourcen, die wir Ihnen verlinken, zu komplementieren oder spezifische Fragen zu klären, empfiehlt sich immer eine Internet-Suche. Die Fähigkeit, einer Suchmaschine die richtigen Fragen zu stellen, ist sehr wertvoll in der angewandten Programmierung.

Inhalt

Einleitung 1
Woche 1 3
Arbeiten mit Java 3
Grundlegende Konzepte von Java 5
Weitere Konzepte 9
Recap 12
Woche 2 14
Vererbung 14
Auf alle Fälle Testfälle 19
Woche 3 26
Der Java Generic Type 26
Von Stapeln und Schlangen – Eine kurze Wiederholung 28
Collections 29
Die Javadoc 29
Exceptions 30
Schlusswort 32

Woche 1

In der ersten Woche beschäftigen wir uns damit, wie Sie mit Java arbeiten können, und erklären die ersten Grundlagen. Neben wichtigen Konzepten von Java wie Klassen, Methoden und objektorientierter Programmierung gehen wir auch auf Arrays und Strings ein und wie diese sich in Java von den Konzepten in C unterscheiden.

Arbeiten mit Java

Wir empfehlen die Arbeit mit einer IDE (Integrated Development Platform) und nutzen für "Algorithmen und Datenstrukturen" IntelliJ IDEA von Jetbrains:
https://www.jetbrains.com/de-de/idea/

Entwicklungsumgebungen wie IDEA helfen dabei, nicht den Überblick über ein Projekt zu verlieren. Außerdem helfen sie gerade am Anfang des Lernprozesses, sich schneller mit einer neuen Sprache vertraut zu machen. Sie sind nicht gezwungen, eine IDE oder IDEA zu nutzen, wir bieten allerdings nur Support für IDEA an.

IDEA installieren

Die Community-Edition von IDEA ist Open-Source und kostenlos und kann unter dem oben genannten Link heruntergeladen werden. Das Programm kann entweder mit dem Jetbrains-Installer oder eigenständig installiert werden, folgen Sie dazu einfach den Anweisungen hier:
https://www.jetbrains.com/help/idea/installation-guide.html

Java installieren

Auf vielen Geräten existiert bereits eine Version von Java, Sie benötigen allerdings ein Java Development Kit (JDK), um lokal Java-Code zu kompilieren. IDEA benötigt ebenfalls Zugriff auf das JDK um einsatzbereit zu sein. Daher empfehlen wir, das neueste JDK der Java Standard Edition, JDK 14 herunterzuladen und zu installieren, welches Sie hier finden:
https://www.oracle.com/java/technologies/javase-downloads.html

Das JDK enthält ebenfalls das entsprechende Java Runtime Environment. Es muss also nicht extra installiert werden falls auf Ihrem Gerät vorher kein Java installiert war. Die Installation verläuft je nach Betriebssystem leicht unterschiedlich. Eine detaillierte Anleitung für jedes gängige Betriebssystem ist hier zu finden:
https://docs.oracle.com/en/java/javase/14/install/overview-jdk-installation.html#GUID-8677A77F-231A-40F7-98B9-1FD0B48C346A

In IDEA Code schreiben und Kompilieren

Sobald IDEA und das JDK installiert sind, können Sie in IDEA nahtlos Code schreiben, kompilieren und ausführen. IDEs nutzen in der Regel Projekte, um Dateien und Code semantisch zu bündeln. Um sich mit Java vertraut zu machen, fangen wir mit einem einfachen Beispiel an.

Aufgabe

- Öffnen Sie IntelliJ IDEA und wählen Sie auf dem Start-Screen „New Project“ aus oder gehen Sie im Menü über File/New/Project.
- Wählen Sie dann auf der linken Seite „Java“ aus. Wenn das JDK schon in IntelliJ definiert ist, wählen Sie es aus der „Project SDK“-Liste aus, ansonsten wählen Sie „Add JDK…“, um das installierte JDK auszuwählen.
- Klicken Sie zweimal auf „next“, um den Namen und den Ordner für ihr Projekt auszuwählen, nenne Sie das Projekt „HelloWorld“ und klicken Sie auf „finish“.
- Auf der linken Seite können Sie nun Ihr Projekt ausklappen. Gehen Sie dort in den src-Ordner
- Gehen Sie nun im Menü über File/New/Java Class und nennen Sie ihre neue Klasse „HelloWorld“.

Sie sollten eine Klasse mit dem folgenden vorgegebenen Code erhalten:

```
public class HelloWorld {
}
```

Schreiben Sie nun zwischen die geschweiften Klammern die main-Methode wie folgt:

```
public static void main(String[] args){
}
```

Schreiben Sie dann in diese Methode die folgende Zeile. Tippen Sie die Zeile ab, damit Sie sehen, was passiert.

```
System.out.println("Hello World!");
```

Sie haben sicher bemerkt, dass IntelliJ Ihnen Vorschläge zum Code macht. Wenn Sie beispielsweise `System.out.p` tippen, können Sie einfach „Enter“ drücken und der Befehl wird vervollständigt. Und es kommt noch besser! Wenn Sie `sout` eintippen und „Enter“ drücken, wird der komplette Befehl eingefügt.
Ihr Code sollte nun so aussehen:

```
public class HelloWorld {
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

Klicken Sie nun oben im Menü auf Run/Run… und wählen sie „Hello World“ aus. Alternativ können Sie auch neben dem Code auf das grüne Dreieck klicken und so die komplette Klasse oder die `main`-Methode ausführen. Das Programm kompiliert kurz, dann sollte sich unten ein Fenster öffnen, indem Ihr gewünschter Text ausgegeben wird.

Grundlegende Konzepte von Java

Klassen und Methoden

Nachdem Sie in der vorherigen Lektion Ihren ersten Code geschrieben und ausgeführt haben, möchten wir nun anhand dieses Codes verstehen, was die Kernkonzepte von Java sind.

In dem kurzen Stück Code finden sich bereits mehrere dieser Konzepte wieder. Wie in C befindet sich ausführbarer Code immer in einer Funktion, welche in Java “Methode” genannt wird. Die einzige Methode, die Sie bis jetzt sehen können, ist die `main`-Methode. Wie in C und den meisten gängigen Programmiersprachen wird die `main`-Methode beim Starten des Programms ausgeführt. Diese `main`-Methode steht innerhalb der Klasse `HelloWorld`. Klassen sind das zentrale Konzept in Java.

Wir wollen hier einen kurzen Überblick über die wichtigsten Begriffe geben:

Variable
Eine Variable hat – wie der Name schon sagt – einen variablen Wert. In ihr können je nach Typ der Variable Zahlen, Wörter oder Buchstaben gespeichert werden (Dazu später mehr). Eine Variable sollte einen passenden Namen bekommen, der beschreibt, was in ihr gespeichert werden soll.

Deklarieren
Beim Deklarieren wird nur festgelegt, dass es eine Variable von einem bestimmten Typ mit einem bestimmten Namen gibt (Erst Typ/Klasse, dann Name).

```
int zahl;
```

Initialisieren
Beim Initialisieren wird der Variablen ein Wert (oder Objekt) zugewiesen.

```
zahl = 5;       //Initialisierung
int zahl = 5;   //Deklaration und Initialisierung in einem
```

Klasse
Um Unmengen von Code sinnvoll zu strukturieren, wird er in Klassen gegliedert. Eine Klasse ist dabei oft eine eigene Datei, der Klassenname beginnt üblicherweise mit einem Großbuchstaben und sollte beschreiben, was für Code in der Klasse enthalten ist. Unsere `HelloWorld`-Klasse kann beispielsweise nichts anderes, als „Hello World!“ auszugeben,

daher ist der Name sehr passend. Später werden Sie auch Objekte von Klassen wie z.B. `Pferd` erzeugen. Die Klasse enthält dann alle Eigenschaften und Fähigkeiten, die ein Pferd haben kann. Eine Klasse beginnt und endet mit geschweiften Klammern und Java verzeiht es nicht, wenn Sie sie vergessen!

```
public class Pferd {
    //Attribute und Methoden
}
```

Methode

Methoden werden wie unsere `main`-Methode innerhalb von Klassen definiert. Die `main`-Methode ist dabei ein spezieller Fall, auf den wir später genauer eingehen werden. Eine Methode tut immer etwas, wenn sie ausgeführt wird, sie verändert den Wert einer Variablen, gibt etwas auf dem Bildschirm aus oder berechnet etwas. Der Methodenname sollte beschreiben, was die Methode tut und beginnt üblicherweise mit einem Kleinbuchstaben. Auch Methoden beginnen und enden mit geschweiften Klammern.

```
public static int sum(int a, int b){
    return a+b;
}
```

Auf die Keywords public und static gehen wir später genauer ein. `int` ist der Rückgabetyp und gibt an, dass die Methode eine ganze Zahl als Ergebnis zurückgibt. Der Name der Methode ist `sum` und sie bekommt zwei ganze Zahlen `a` und `b` übergeben, aus denen sie die Summe berechnet und zurückgibt. Die Parameter in Klammern müssen immer mit einem Datentyp (hier `int`) versehen werden.

Objekt

Wir hatten eben das Beispiel der Klasse `Pferd`. Wenn Sie diese Klasse schreiben würden, könnte sie Variablen wie `geschwindigkeit`, `groesse`, `gewicht`, `farbe` sowie einige Methoden enthalten. Die Werte für diese Variablen wären aber in der Klasse noch nicht vergeben. Die Klasse ist in diesem Fall also lediglich ein unausgefülltes Formular, in das noch konkrete Werte eingetragen werden müssen. Erzeugen Sie nun ein Objekt der Klasse Pferd, weisen Sie den Variablen Werte zu. So können während der Laufzeit ihres Programms viele Objekte der Klasse `Pferd` existieren, die aber alle unterschiedliche Eigenschaften haben.

Objektorientierte Sprache

Java ist eine objektorientierte Sprache, das heißt, sie ist darauf ausgelegt mit den beschriebenen Konzepten, von Klassen, Methoden und eben Objekten zu arbeiten. Das macht es, wie man am Beispiel der Klasse `Pferd` sieht, auch für Laien einfach, sich in den Code hineinzudenken, da Konzepte aus der realen Welt sich gut in Java abbilden lassen.

Punkt und Semikolon

Der Punkt dient in Java dem Unterteilen von Klassen bzw. Objekten und Methoden bzw. Variablen. Die Variable `geschwindigkeit` eines `Pferd`-Objektes `kleinerOnkel` kann über `kleinerOnkel.geschwindigkeit` abgerufen werden. Das Semikolon schließt wie in den Beispielen gesehen einen Befehl ab.

Weitere Begriffe wie „Package“, „import“ und „@Test“ können Interessierte in dieser Einführung in Java nachlesen:
https://marcus-biel.com/java-course-introduction/

Neben der `main`-Methode wird in Ihrem kleinen Programm noch eine zweite Methode aufgerufen, auch wenn Sie selbst keinen Zugriff auf den Inhalt haben. Die `println`-Methode, die dafür sorgt, dass Ihr Text auf der Konsole ausgegeben wird.

Wenn Sie in IDEA einen Befehl Stück für Stück eintippen, schlägt die IDE Ihnen nach jedem Punkt Optionen vor, die Sie der Klassenstruktur entnimmt, auf die Sie mit dem Befehl zugreifen wollen. Daher können wir sehen, dass die `println`-Methode der `PrintStream`-Klasse angehört und auf einem Objekt dieser Klasse aufgerufen werden muss. Die Klasse `System` hat ein Attribut (auch oft Feld genannt) dieser Klasse namens `out`.
Daraus setzt sich dann folgender Aufruf zusammen: `System.out.println()`. Wir greifen auf das Feld `out` der Klasse `System` zu, und wollen die Methode `println()` der `PrintStream`-Klasse auf dem Objekt in diesem Feld ausführen.

Aufgabe

Schreiben Sie nun in Ihrer `HelloWorld`-Klasse eine neue Methode `public static void mehrfachAusgabe(int anzahl)`, die den Text `anzahl` mal auf der Konsole ausgibt, und rufen Sie diese in der `main`-Methode mit einem beliebigen Wert für `anzahl` auf. (Die Keywords `public`, `static` und `void`, die Sie bereits in der `main`-Methode gesehen haben, werden später genauer erklärt.)

Objekte und Konstruktoren

Objekte sind spezifische Instanzen einer Klasse. Das vom Fließband laufende Auto ist zum Beispiel eine Instanz (ein Objekt) seines Modells (der Klasse). Unsere `HelloWorld`-Klasse existiert bisher nur als Bauplan. Kommen wir nochmal zu unserem Pferde-Beispiel zurück. So könnte eine solche `Pferd`-Klasse aussehen.

```
public class Pferd {
    public int geschwindigkeit;
    public String farbe;

    //Parameterloser Konstruktor
    public Pferd(){
        this.geschwindigkeit = 0;
        this.farbe = "schwarz";
    }

    //Parametrisierter Konstruktor
    public Pferd(int geschwindigkeit, String farbe){
        this.geschwindigkeit = geschwindigkeit;
        this.farbe = farbe;
    }
```

```
    public static void wiehern(){
        System.out.println("Wiieeher, schnaub");
    }

    public void springen(){
        System.out.println("Hui!");
    }
}
```

Die Klasse enthält zwei Eigenschaften eines Pferdes, die in den Variablen `geschwindigkeit` und `farbe` gespeichert werden können. Danach folgen zwei Methoden bei denen Ihnen vielleicht eine Besonderheit aufgefallen ist: Sie verfügen über keinen Rückgabetyp, auch nicht `void`. Außerdem tragen sie den gleichen Namen wie die Klasse. Diese Methoden sind Konstruktoren. Ein neues Objekt der Klasse `Pferd` kann über einen dieser Konstruktoren aus einer anderen Klasse erstellt werden. Ein solcher Aufruf könnte so aussehen:

```
Pferd kleinerOnkel = new Pferd(60, "weiß");
```

Die Klasse verfügt über einen parametrisierten und einen parameterlosen Konstruktor. Über den parametrisierten kann wie in der Beispiel-Zeile ein Pferd mit gewünschten Eigenschaften erzeugt werden, wird der parameterlose aufgerufen, bekommt das Pferd die vordefinierten Eigenschaften. Wenn Sie keinen parameterlosen Konstruktor schreiben, dieser aber trotzdem aufgerufen wird, bekommen die Variablen Standardwerte zugeteilt. Für einen `int` wäre das 0, für einen `String` null, was bedeutet, dass dieser nicht definiert wurde.

Sie sind in den Konstruktoren vielleicht über das `this` gestolpert. Es wird immer dann notwendig, wenn unklar ist, ob die übergebene Variable oder die Klassenvariable gemeint ist. So bei `this.geschwindigkeit = geschwindigkeit`. Im parameterlosen Konstruktor wäre es eigentlich nicht nötig, da dort keine zwei Variablen mit gleichem Namen auftreten, der Übersichtlichkeit halber hilft es aber manchmal, es trotzdem zu schreiben.

Weiter unten in der Klasse finden Sie die Methoden `wiehern` und `springen`. Überlegen Sie, welche von diesen Methoden Sie einfach aus der `Pferd`-Klasse aufrufen können und für welche Sie ein `Pferd`-Objekt benötigen.

Aufgabe

Nun wollen wir alle gelernten Grundkonzepte auf unseren Code anwenden. Schreiben Sie einen Konstruktor für Ihre `HelloWorld`-Klasse und erstellen Sie eine Instanz Ihrer Klasse in der `main`.

Schreiben Sie außerdem eine weitere Methode für die Klasse welche die Funktionalität von `mehrfachAusgabe()` implementiert, allerdings nicht das `static`-Keyword verwendet. Werden Sie kreativ. Sie können Attribute zu Ihrer Klasse hinzufügen, welche mit dem Konstruktor gesetzt werden können, unter anderem z.B. den Wert von `anzahl`.

Diese Methode können Sie nun auf Ihrem erstellten Objekt aufrufen. Weitere Attribute können Sie dann in den Text einfließen lassen, der von Ihrer Methode ausgegeben wird, um zum Beispiel Ihren Namen hinzuzufügen. Experimentieren Sie ruhig etwas.

Mit dem "+"-Operator können Sie Strings in Java aneinanderhängen, im Fachjargon "konkatenieren" genannt. Java arbeitet hier mit dem Prinzip des Overloading (ein Symbol oder Methodenname kann mehrere Bedeutungen haben) und erkennt im Kontext von Strings, dass diese im Gegensatz zu beispielsweise Integern nicht addiert, sondern konkateniert werden sollen.

```
public static void main(String[] args){
    int addition = 4+4;
    String konkatenation = "4" + "4";

    System.out.println("Eine Addition:\t\t" + addition);
    System.out.println("Eine Konkatenation:\t" + konkatenation);
}
```

Ausgabe:

```
Eine Addition:        8
Eine Konkatenation:   44
```

Weitere Konzepte

Bis jetzt haben wir einige Begriffe und Code-Schnipsel bewusst ignoriert. Mithilfe der Autovervollständigung haben Sie syntaktisch korrekten Code schreiben können, ohne alle Nuancen zu verstehen. Jetzt betrachten wir erneut die erste Methode, die Sie geschrieben haben, und befassen uns mit den Details.

Ein zweiter Blick auf die Main-Methode

```
public static void main(String[] args){
    //Your Code
}
```

Die `main`-Methode besteht aus einigen Schlüsselwörtern, die wir uns noch nicht genau angesehen haben.

Schlüsselwort	Bedeutung
`public`	Die Methode kann auch von außerhalb ihrer eigenen Klasse aufgerufen werden.
`static`	Die Methode kann aufgerufen werden, ohne dass ein Objekt der Klasse erzeugt werden muss, in der sie sich befindet.
`void`	Die Methode gibt nichts zurück (Eine Methode, die zwei Zahlen addiert, z.B. würde eine Zahl zurückgeben (nämlich das Ergebnis)).
`main`	Der Name der Methode
`String[] args`	Die Methode bekommt ein Array von Strings als Parameter übergeben

Wenn Ihr Code ausgeführt werden soll, benötigt Java einen Punkt, von dem aus das Programm startet. Dieser Ausgangspunkt ist die `main`-Methode. Sämtliche anderen Methoden werden nicht automatisch ausgeführt und müssen aus der `main` aufgerufen werden, sie steuert also den Ablauf des Programms. Dafür muss die Signatur der Methode exakt `public static void main` sein, da das Java Runtime Environment die Methode sonst nicht erkennt.

Keywords

Das erste Keyword `public` bestimmt die Sichtbarkeit unserer Methode und kann analog so auch auf Klassen und Klassenfelder angewandt werden. Hier eine Übersicht der Modifizierer für die Sichtbarkeit.

Modifizierer	Bedeutung
default	Wird nicht hingeschrieben, kein Modifizierer, Zugriff aus Klassen im selben Packet
`public`	Zugriff aus jeder anderen Klasse
`private`	Zugriff nur aus der eigenen Klasse
`protected`	Zugriff aus Klassen im selben Packet oder Klassen, die von der Klasse erben

Das zweite Keyword, `static`, teilt dem Compiler mit in welchem Bezug die Methode oder Variable zu der Klasse steht. Wenn Sie Methoden mit dem Keyword `static` versehen, müssen Sie also nicht auf Instanzen Ihrer Klasse aufgerufen werden. Vergleichen Sie die beiden Methoden, die Sie geschrieben haben. Fallen Ihnen Unterschiede auf?

Für unsere instanzierte Methode ist das `this`-Keyword ein Platzhalter für das Objekt, das während der Runtime die Methode ausführt. Auf statischen Methoden existiert kein Referenz-objekt weshalb unser Compiler hier einen Fehler anstreicht.

Wenn Sie Code schreiben, werden Sie sowohl statische als auch Objekt-Methoden verwenden, behalten Sie also beide Möglichkeiten im Kopf.

Datentypen

Java ist eine typisierte Sprache und benötigt daher zur Compile-Zeit konkrete Datentypen für jede Variable. Jede Methode benötigt einen Rückgabewert und daher auch den Datentyp, den diese Variable haben soll. (Eine Ausnahme davon sind Generics, welche wir in Woche 3 besprechen werden.)

```
public int addition(int summandA, int summandB){
    //Inhalt...(Methodenrumpf)
}
```

Unsere `main` hat den Rückgabetyp `void` und deklariert damit, dass beim `return` keine Variable übergeben wird. Die Java Runtime sucht nach einer `main`-Methode mit exakt dieser Signatur, ändern Sie also dort nicht den Rückgabetyp.

In Java existieren die weit verbreiteten Standard-Typen `byte`, `short`, `int` und `long` für Integer-Zahlen, `float` und `double` für Fließkomma-Zahlen sowie `boolean` für Wahrheitswerte und `char` für Unicode-Werte (einzelne Zeichen).
In der offiziellen Java Dokumentation finden Sie einen genaueren Überblick über alle primitiven Daten-typen und deren Konfiguration:
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Strings, Arrays und der Rest

Darüber hinaus sind alle komplexen Datentypen Klassen. Ein String ist ein Objekt der Klasse `String`, ein Array ein Objekt der Klasse `Array`. Alle Methoden, die auf Strings und Arrays ausgeführt werden können, kommen direkt aus deren zugehörigen Klassen.

Unsere `main`-Methode hat ein Array von Strings als Übergabeparameter. Dies sind die Argumente die dem Programm von außen, also zum Beispiel über die Konsole, mitgegeben werden können. Wie Arrays grundlegend funktionieren, wissen Sie bereits aus IntroProg. Hier eine kleine Auffrischung für Java:

```
//autos ist ein neues Array vom Typ Auto mit Länge 23
Auto[] autos = new Auto[23];

//Ein neu erstelltes Auto an Stelle 5 ins Array einfügen
autos[5] = new Auto();

//autochen soll auf dasselbe Auto verweisen
// wie das Array an der Stelle 17
Auto autochen = autos[17];
```

```
//auchAutos wird eine neue Referenz auf dasselbe Array
Auto[] auchAutos = autos;
```

Weitere Details und Beispiele können Sie hier nachlesen:
https://marcus-biel.com/java-arrays-enums/

Objekte sind immer Referenz-Typen. Wenn Objekte in einem Array gespeichert werden, wird ihre Referenz gespeichert. Die Variable, die das Array hält, speichert selbst nur eine Referenz auf das Array. Insofern weisen Objekte hier eine gewisse Ähnlichkeit mit structs aus C auf. Das heißt auch, dass Sie Arrays genau wie andere komplexe Objekte nicht einfach kopieren können! Für Arrays eigenen sich hier for-Schleifen, um Inhalte einzeln in ein neues Array zu übertragen.
Das Keyword `new` entspricht in etwa einem `malloc()` in C, nur dass Java komplett die Speicher- und Pointer-Verwaltung übernimmt und Speicher somit automatisch wieder freigibt (in Lektüre wird diese Funktion mit “Garbage Collection” betitelt).
Seien Sie sich also bewusst, dass beim Arbeiten mit Variablen komplexen Datentyps die Referenz-Typ Regeln gelten.

Ein spezieller for-Loop

In den verlinkten Lektionen ist Ihnen vielleicht bereits eine besondere Form des for-Loops aufgefallen. Der sogenannte for-each-Loop iteriert über alle Objekte einer Klasse, die sich in einem gegebenen Container befinden.

```
for(ClassY y: arrayA){
    System.out.print(y);
}
```

Kann gelesen werden als:
„Für jedes Objekt `y` der Klasse `ClassY` in `arrayA`, gib `y` aus.“

Die for-Schleife iteriert selbstständig durch das gegebene Array oder jeden anderen Container in Java, der diese Form der Iteration unterstützt (mehr dazu in den späteren Kapiteln), und für alle Objekte, die die gegebene Bedingung erfüllen, wird der Schleifenkörper ausgeführt.

Die Unterschiede zwischen for- und for-each-loop können Sie hier noch einmal genauer nachlesen:
https://marcus-biel.com/for-each-loop/

Recap

Wir kennen nun Methoden, die Werkzeuge, mit denen unserer Code Arbeit verrichten kann, sowie Klassen, die sowohl Baupläne sind als auch eine Bündelung und Abgrenzung von

Konzepten darstellen, und Objekte, welche die “anfassbaren” und manipulierbaren Instanzen unserer Klassen sind.

Objekte sind der Grund auf dem Objekt-orientierte Sprachen stehen. Sie sollten also komfortabel im Umgang mit ihnen sein und nicht davor zurückschrecken sie einzusetzen. Fühlen Sie sich aber nicht gezwungen, alles mit Hilfe von Klassen-Instanzen zu lösen.

Bevor wir uns von unserer `HelloWorld`-Klasse verabschieden, wollen wir ihr noch einen letzten Schliff verleihen. Die allgemein verbreitete Sprache in der Informatik ist Englisch, wodurch andere Sprachen oft zu kurz kommen. Dem wollen wir entgegenwirken!

Aufgabe

Schreiben Sie eine Methode `sayHello()` ohne Rückgabewert und ohne Parameter, welche in der Lage ist “Hello World!” in unterschiedlichen Sprachen wiederzugeben (wie viele ist Ihnen überlassen) Eine Möglichkeit den Code dafür zu schreiben ist ein `switch`-Statement. (Hier ist eine Erläuterung der Funktionalität https://marcus-biel.com/java-switch-statement/).

Eventuell müssen Sie Attribute zu Ihrer Klasse hinzufügen. Anstatt Ihren bestehenden Konstruktor zu verändern schreiben Sie stattdessen einen neuen größeren Konstruktor mit anderer Signatur. Eine Klasse kann beliebig viele Attribute und Konstruktoren haben, stellen Sie aber sicher das Sie oder andere nicht den Überblick verlieren. Kommentare helfen hier.

Erstellen Sie außerdem ein kleines Array, in dem für jede Sprache ein typischer Name steht, und lassen Sie ihr Programm sich selbst vorstellen. Natürlich mit den passenden Namen für jede Sprache.

Wenn Sie möchten, können Sie auch ein zweidimensionales Array erstellen, und eine Methode schreiben, die im Konstruktor aufgerufen wird und einen von mehreren Namen auswählt, der in dem Array steht.

Sobald Sie fertig sind, erstellen Sie ein Objekt mit dem neuen Konstruktor und führen Sie auf dem Objekt die `sayHello`-Methode aus.

Fertig!

Bedenken Sie, das neue Objekt ist von der gleichen Klasse und hat Zugriff auf alle Attribute, auch welche diese im Konstruktor nicht gesetzt wurden. Die beiden unterschiedlichen Konstruktoren erzeugen also Objekte derselben Klasse, nur mit unterschiedlichen Start-eigenschaften. Wie zum Beispiel Autos, die mit und ohne Klimatisierung verkauft werden. Bei beiden kann die Klimaanlage nachträglich umgerüstet werden.

Hola mundo, mi nombre es Robin! Ciao mondo, mi chiamo Robin! Privet mir, menya zovut Robin!

Woche 2

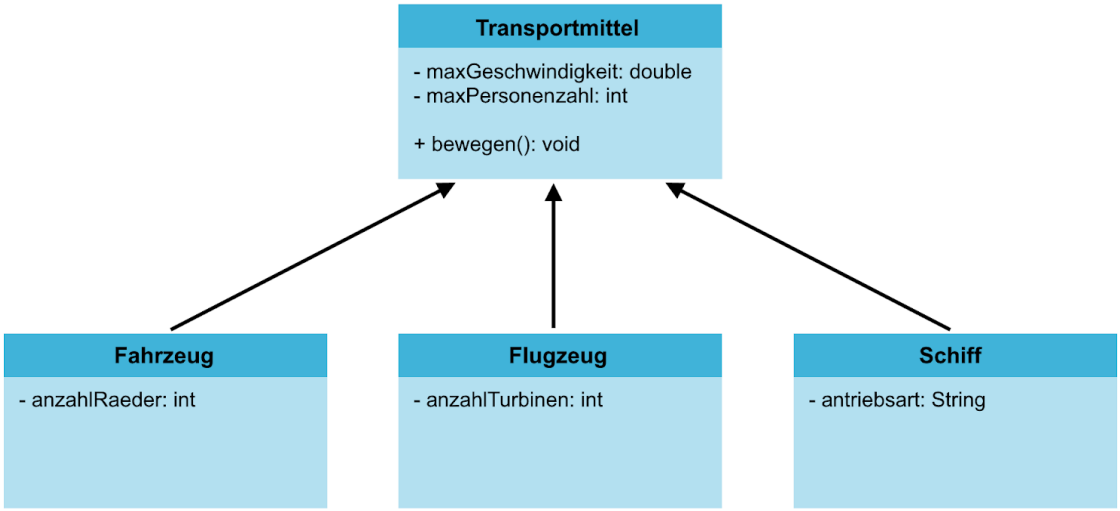
In dieser Woche behandeln wir Vererbung, abstrakte Klassen und Interfaces, die wichtig für die objektorientierte Programmierung sind. Außerdem beschäftigen wir uns damit, wie Sie ihren Code testen können.

Vererbung

Wie Sie in Woche 1 gelernt haben, sind Objekte in Java Instanzen von Klassen. Das Konzept der Vererbung ermöglicht es, diese Klassen zueinander in Beziehung zu setzen.

Super- und Subklassen

Bei der Vererbung findet in Java von der Superklasse (der übergeordneten Klasse) zur Subklasse (der untergeordneten Klasse) immer eine Spezialisierung statt. So kann z.B. die Klasse `Transportmittel` die Superklasse sein, von der die Klassen `Schiff`, `Fahrzeug` und `Flugzeug` erben. Von der Klasse `Fahrzeug` wiederum könnten die Klassen `Zug` und `Fahrrad` erben und so weiter.



Dabei werden bei jeder Spezialisierung alle Attribute und Methoden der Superklasse geerbt (jedes `Fahrzeug`, `Flugzeug`, `Schiff` hat eine `maxGeschwindigkeit`), es können aber neue Attribute und Methoden hinzukommen oder alte überschrieben werden. Wichtig ist, dass jeweils nur von einer Superklasse geerbt werden kann. Hier ein Beispiel für die Klasse Flugzeug:

```
public class Flugzeug extends Transportmittel{
    int anzahlTurbinen;
}
```

Das Schlüsselwort `extends` kennzeichnet hier, dass die Klasse `Flugzeug` eine Subklasse der Klasse `Transportmittel` ist. Was an Attributen und Methoden bereits in `Transportmittel` vorhanden ist, muss in `Flugzeug` nicht nochmal geschrieben werden. Lediglich das zusätzliche Attribut `anzahlTurbinen` muss ergänzt werden.

Aufgabe

Schreiben Sie die Klasse `Transportmittel` mit den Attributen aus dem Diagramm und einem parametrisierten Konstruktor. Lassen Sie die Methode `bewegen()` zunächst weg.

Schreiben Sie die Klasse `Fahrzeug`, die von `Transportmittel` erbt.

Methoden, die bereits in einer Superklasse vorkommen, können in Subklassen auch überschrieben werden, wenn sie dort eine veränderte Funktion erfüllen sollen. Die Methode wird dann einfach mit der gleichen Signatur wie in der Superklasse geschrieben, nur der Inhalt wird verändert. Zum Beispiel so für die Methode `bewegen()` in `Flugzeug`:

```
public void bewegen(){
    System.out.println("Ich fliiiieeege!!!");
}
```

Wenn Sie eine Methode in einer Subklasse nicht anpassen wollen, erwähnen Sie sie einfach nicht, dann wird sie unverändert aus der Superklasse übernommen.

Wie bei den Attributen können Sie auch im Konstruktor einer Subklasse faul sein und sich Code sparen, indem Sie den Superkonstruktor (den Konstruktor der Superklasse) aufrufen, da in diesem bereits einige Attribute initialisiert. Der Aufruf erfolgt über `super(parameterA, parameterB…)`. Allerdings müssen Sie dabei beachten, dass der Aufruf das Erste sein muss, was in Ihrem Konstruktor passiert! Gehen wir davon aus, dass die Klasse `Transportmittel` über einen parametrisierten Konstruktor verfügt, der alle Attribute initialisiert. Dann könnte der Konstruktor für `Flugzeug` wie folgt aussehen:

```
public Flugzeug(double maxGeschwindigkeit, int maxPersonenzahl,
int anzahlTurbinen){

    super(maxGeschwindigkeit, maxPersonenzahl);
    this.anzahlTurbinen = anzahlTurbinen;
}
```

Es muss also nur noch das neue Attribut `anzahlTurbinen` „per Hand“ initialisiert werden, den Rest erledigt der Superkonstruktor.

Schreiben Sie nach diesem Vorbild nun auch einen Konstruktor für `Fahrzeug`.

Polymorphie

“Polymorphie oder Polymorphismus (griechisch für Vielgestaltigkeit) ist ein Konzept in der objektorientierten Programmierung, das ermöglicht, dass ein **Bezeichner** abhängig von seiner Verwendung **Objekte unterschiedlichen Datentyps** annimmt.”
- Wikipedia

Keine Angst, Sie müssen diese Definition nicht sofort verstehen. Wir erklären sie an unserem Beispiel.

Aufgabe

Schreiben Sie dazu zunächst die Klasse `Flugzeug`. Sie können dabei viel aus den Beispielen übernehmen.

Erstellen Sie dann eine weitere Klasse (z.B. `TestTransportmittel`) mit einer main-Methode mit dem folgenden Code:

```
Transportmittel[] array = new Transportmittel[2];
array[0] = new Fahrzeug(60.5, 2, 2);
array[1] = new Flugzeug(1185, 853, 4);
```

Wie Sie sehen, können Sie Objekte der Klassen `Fahrzeug` und `Flugzeug` in ein Array vom Typ `Transportmittel` speichern, da Transportmittel die Superklasse der beiden ist.

Versuchen Sie nun, sich `array[0].anzahlRaeder` auf der Konsole ausgeben zu lassen.

Die Fehlermeldung, die Sie bekommen, kommt daher, dass `array[0]` nach außen ein Objekt vom Typ `Transportmittel` ist, und ein Transportmittel hat nicht zwingend eine Anzahl Räder. Um das Objekt als Fahrzeug zu verwenden, müssen Sie casten:

```
((Fahrzeug)array[0]).anzahlRaeder
```

gibt Ihnen die Anzahl der Räder von `array[0]` aus. Der Cast passiert, indem der gewünschte Datentyp in Klammern vor die Variable geschrieben wird. Die äußeren Klammern sind hier notwendig, um sicherzustellen, dass erst gecastet und dann das Attribut `anzahlRaeder` abgefragt wird.

Wir sehen also, dass der Bezeichner `array[0]` je nach Verwendung entweder auf ein Objekt vom Typ `Transportmittel` oder auf ein Objekt vom `Fahrzeug` verweisen kann. Das ist Polymorphie.

Beim Casten muss jedoch beachtet werden, dass dies nicht beliebig funktioniert. `array[1]` könnte nicht zum Typ `Fahrzeug` gecastet werden, weil es ein Flugzeug ist. Um zu überprüfen, ob ein Cast möglich ist, verwenden wir den Vergleichsoperator `instanceof`. Es liefert true, wenn das Objekt vom Typ der Klasse oder vom Typ einer Subklasse der abgefragten Klasse.

```
array[1] instanceof Flugzeug           => true
array[1] instanceof Fahrzeug           => false
array[1] instanceof Transportmittel    => true
```

Abstrakte Klassen

Wir beschäftigen uns nun mit der Methode `bewegen()`. Diese soll die Art der Bewegung auf der Konsole ausgeben, also zum Beispiel für ein Objekt der Klasse `Schiff` “ich schwimme”. Wie im Vererbungsdiagramm weiter oben zu sehen ist, wird die Methode schon in der Klasse `Transportmittel` deklariert. Es ist aber schwierig, für ein allgemeines Transportmittel zu sagen, wie es sich bewegt. Java bietet hier die Möglichkeit, `bewegen()` als **abstrakte Methode** zu schreiben. Die Methode wird in `Transportmittel` nur deklariert (es wird nur der Methodenkopf geschrieben) und erst in den Subklassen implementiert (der Methodenrumpf wird geschrieben). So ist sichergestellt, dass jede Subklasse von Transportmittel über die Methode `bewegen()` verfügt, was sie genau tut, kann sich jedoch unterscheiden. Die Deklaration einer abstrakten Methode kann wie folgt aussehen:

```
public abstract void methode();
```

Sobald eine Klasse mindestens eine abstrakte Methode enthält, ist auch die Klasse selbst abstrakt und braucht ebenfalls das Schlüsselwort abstract:

```
public abstract Klasse{}
```

Aufgabe

Schreiben Sie in der Klasse `Transportmittel` die abstrakte Methode `bewegen()` und machen Sie `Transportmittel` zu einer abstrakten Klasse.

Abstrakte Klassen haben auch in ihrer Verwendung einen Unterschied. Wenn Sie nun versuchen würden, ein Objekt vom Typ `Transportmittel` zu erzeugen, würden Sie eine Fehlermeldung erhalten. Ein `Transportmittel`-Objekt hätte mit `bewegen()` eine quasi “unfertige” Methode, die nicht verwendet werden kann. Deshalb können von abstrakten Klassen generell keine Objekte erzeugt werden.
Klassen, die von abstrakten Klassen erben, sind selbst abstrakt, es sei denn, Sie implementieren alle Methoden, so dass keine abstrakten Methoden mehr übrig bleiben.

Implementieren Sie `bewegen()` in `Fahrzeug` und `Flugzeug` und testen Sie, ob ihr Code funktioniert.

Interfaces

Interfaces sind Schnittstellen, über die bestimmte Vorgaben für Klassen gemacht werden können. Ein Interface besitzt nur abstrakte Methoden, keine vollständig implementierten Methoden oder Attribute, deshalb können keine Objekte von Interfaces erzeugt werden. Da alle Methoden abstrakt sind, wird das Schlüsselwort `abstract` hier nicht benötigt. So könnte ein Interface aussehen:

```
public interface Schnell {
    double getMaxTempo();
    boolean istSchneller(Schnell anderes);
}
```

Das Interface `Schnell` ermöglicht es, eine Klasse, die `Schnell` implementiert, mit einer anderen Klasse, die `Schnell` implementiert zu vergleichen. Die Methoden eines Interfaces sind automatisch `public`, sofern wie hier kein Sichtbarkeitsmodifizierer geschrieben wird. Sie können `public` jedoch auch davor schreiben, wenn Ihnen das übersichtlicher erscheint.

Eine Klasse, die ein Interface implementiert, muss alle Methoden des Interfaces implementieren. Der Vorteil gegenüber der Vererbung ist, dass die Klassen hier nicht verwandt sein müssen. Die Klasse `Fahrzeug` kann das Interface genauso implementieren wie die Klasse `Gepard` und wir können so ein Transportmittel mit einem Tier vergleichen. Außerdem können mehrere Interfaces implementiert werden, während nur von einer Klasse geerbt werden kann. Eine Klasse kann Interfaces wie folgt implementieren:

```
public class Fahrzeug extends Transportmittel implements
Interface1, Interface2, Interface3, Interface4{}
```

Das Schlüsselwort ist hier `implements` statt `extends`, falls mehrere Interfaces implementiert werden, werden sie mit Komma getrennt.

Ein weiteres Beispiel können Sie sich hier ansehen:
https://www.learnjavaonline.org/en/Interfaces

Aufgabe

Kopieren Sie das Interface `Schnell` und speichern Sie es in ihrem Ordner.

Schreiben Sie die Klasse `Gepard`. Diese muss von keiner Klasse erben, sie soll aber ein Attribut `maxGeschwindigkeit` haben und das Interface `Schnell` implementieren. Die Methode `getMaxTempo()` soll dabei die maximale Geschwindigkeit zurückgeben, die Methode `istSchneller(Schnell anderes)` soll mit der `getMaxTempo`-Methode vergleichen, welches Objekt schneller ist.

Binden Sie das Interface `Schnell` auch in Fahrzeug ein, erzeugen Sie ein `Fahrzeug`- und ein `Gepard`-Objekt und vergleichen Sie, welches schneller ist.

Abstrakte Klassen vs. Interfaces

Hier nochmal eine Kleine Zusammenfassung zu den Unterschieden zwischen abstrakten Klassen und Interfaces.

Abstrakte Klassen	Interfaces
`extends`	`implements`
Es kann von maximal einer geerbt werden	Es können mehrere implementiert werden
auch implementierte Methoden und Attribute möglich	Nur abstrakte Methoden (kein `abstract` nötig)
Es können keine Objekte erzeugt werden	Es können keine Objekte erzeugt werden

Recap

Sie wissen nun, wie Klassen mit Hilfe von Vererbung Attribute und Methoden einer Superklasse übernehmen können. Dadurch können Sie Vorgaben für Subklassen machen und sicherstellen, dass wichtige Methoden oder Attribute immer vorhanden sind. Je weiter Sie dabei in der Vererbungshierarchie nach unten gehen, desto spezialisierter werden die Klassen. Sie können Methoden überschreiben und dadurch an einen Kontext anpassen, neue Methoden und Attributen können hinzukommen. Dies ist in der objektorientierten Programmierung ein mächtiges Werkzeug.
Sie können Klassen auch durch eine abstrakte Superklasse Vorgaben machen, ohne dass es von dieser Superklasse überhaupt Objekte geben kann. Beachten Sie dabei, dass die Subklassen in diesem Fall alle Methoden vollständig implementieren müssen, um nicht selbst abstrakt zu sein.
Wo die Vererbung an ihre Grenzen stößt, können Sie Interfaces implementieren, um Klassen, die keine gemeinsame Vererbungshierarchie haben, vergleichbar zu machen oder ihnen bestimmte Methoden vorzuschreiben.

Auf alle Fälle Testfälle

Wenn wir Code schreiben und ihn schlussendlich ausführen, geschehen oft unerwartete Dinge. In der überwältigenden Anzahl von Fällen tut das Programm allerdings nur genau das, was wir ihm mit unserem Code befohlen haben. Wir haben eventuell aber eine Interaktion übersehen oder eine falsche Annahme getroffen.

Jetzt den Fehler zu finden, kann aufwändig werden. Wir können printline-Debugging betreiben oder über einen Debugger wie gdb für C versuchen die Fehlerquelle zu isolieren (IDEA hat auch einen eingebauten Debugger für Java). Aber uns muss bewusst sein, dass der Computer keinerlei Verständnis davon hat, welche Ergebnisse wünschenswert sind. Wir als Programmierer sind dafür verantwortlich zu entscheiden, welche Ergebnisse richtig und welche falsch sind.

Blackbox-Design

Wenn wir ein Programm schreiben, haben wir ein Ziel vor Augen. Das Programm hat einen Sinn, welchen es erfüllen soll, und ein Ergebnis, das es uns oder einem anderen Programm mitteilen kann. Wie es dieses Ziel erreicht und wie lange es dauert, ist uns für den Moment egal. Diese Herangehensweise wird Blackbox genannt, weil wir nicht in die Kiste hinein gucken, die unseren Code darstellt. Uns ist nur wichtig, dass spezifische Inputs von uns erwünschte Outputs erzeugen.

Bevor Sie also Code schreiben, überlegen Sie sich, was die möglichen Eingabetypen sind, und was die Ergebnisse sind, die Ihr Code für jede dieser Eingaben liefern soll. Mit diesen Anforderungen können Sie direkt Tests für Ihr späteres Programm kreieren.

Wenn wir zuerst die Tests schreiben, die die Funktionalität unseres Programms beschreiben und dessen Anforderungen bestimmen, können wir beim Programmieren direkt sehen, ob unsere Lösungen zielführend sind. Außerdem könnten uns beim Schreiben der Tests bereits Lösungen einfallen, die unsere Tests zufrieden stellen. Dieser Ansatz wird “Test-Driven Design” genannt.

Test-Driven Design mit JUnit 5

In Java gibt es mehrere von der Java Community verfasste Testing-Frameworks, die es uns einfach machen, das Testen in unseren normalen Programmierfluss zu integrieren. Wir werden uns in diesem Kurs mit JUnit befassen, da es auch in “Algorithmen und Datenstrukturen” verwendet wird.
JUnit 5 (JUnit Jupiter) ist eine mächtige Testbibliothek, die für uns vor allem zwei wichtige Tools liefert. Die `@Test`-Annotation und Assertions.

Tests können Sie entweder direkt in der Klasse schreiben, die Sie testen möchten, oder eine eigene Klasse verfassen, die nur für das Testen Ihrer Klasse zuständig ist. In jedem Fall fügen Sie einer Methode über dem Methodenkopf `@Test` hinzu, was sie für JUnit als Testfall markiert.

Damit die Methode als Test anerkannt wird, muss sie sowohl `public` sein als auch `void` als Rückgabetyp haben und darf über keinerlei Parameter verfügen.

```
public class ClassYTest{

    @Test
    public void testExample(){

    }
}
```

Die `@Test`-Annotation wird von IDEA noch nicht erkannt, da sie aus der JUnit-Library kommt. IDEA hat allerdings die beiden großen Frameworks JUnit und TestNG bereits vorinstalliert. Sie

müssen JUnit also nur noch importieren. Wie Sie das mit einer handvoll Klicks in IDEA erreichen, können Sie hier lesen:
https://www.jetbrains.com/help/idea/configuring-testing-libraries.html

Nach dem Fix durch IDEA sieht unser Code so aus:

```
import org.junit.jupiter.api.Test;

public class ClassYTest{

    @Test
    public void testExample(){

    }
}
```

Wie schreibt man einen Test

Wir möchten in unserer Klasse `ClassY` eine Methode implementieren, die zwei Integer addiert und das Ergebnis zurückgibt.
Natürlich fangen wir damit an, den Test für diese Methode zu schreiben. Ein guter Test fängt bereits beim Methodenname an. Um klar zu machen, was der Test bezweckt, geben wir ihm einen entsprechenden aussagekräftigen Namen.

Da es der erste Test ist, könnten wir ihn `test1` oder `testA` nennen, was auf der Skala von selbsterklärenden Namen sehr weit unten wäre. Besser ist `testSum` oder `testSumMethod`, es geht allerdings noch selbsterklärender. Eine Namenskonvention für Tests ist die, dass das erwartete Ergebnis im Methodennamen verankert ist. `shouldReturnCorrectSum` oder `shouldReturnSumOfTwoIntegers` sind Namen, bei denen man keine einzelne Codezeile lesen muss, um zu verstehen, was getestet wird.

Allerdings muss man aufpassen, dass die Namen weiterhin mit dem übereinstimmen, was im Test überprüft wird. Hier ein kurzes Beispiel:

```
import org.junit.jupiter.api.Test;
import static org.junit.Assert.*;

public class ClassYTest{
    private ClassY testdummy = new ClassY();

    @Test
    public void shouldReturnCorrectSum(){
        assertEquals(4, testdummy.sumOfTwoInts(2, 2));
    }
    @Test
    public void shouldReturnFour(){
        assertEquals(4, testdummy.sumOfTwoInts(2, 2));
    }
}
```

Welchen dieser identischen Tests halten Sie für besser benannt?
Achten Sie darauf, was ihr Test bewerkstelligt und ob der Test, den Sie schreiben, auch wirklich dem gewählten Namen entspricht und umgekehrt.

Gut gewählte Methodennamen können dabei helfen, die Methode fokussiert zu halten und gleichzeitig nicht den Sinn, den die Methode erfüllen soll, aus den Augen zu verlieren. Der zweite Test ist passender benannt, mit Hilfe des ersten Namens könnten wir aber feststellen, dass wir eventuell zu wenige Testfälle durchgehen, um die Funktionalität gewährleisten zu können.
Aber wie genau funktionieren die oben gezeigten Tests?

Assertions

JUnit nutzt das Konzept von Assertions, Abfragen die eine gewisse Bedingung erwarten und die Methode beenden, falls diese nicht erfüllt ist. Assertion aus dem Englischen bedeutet so viel wie Aussage oder Behauptung.

Wir behaupten, unser Code könne bestimmte Dinge und diese Behauptung wird dann durch unsere Tests auf die Probe gestellt. JUnit hat eine Reihe von unterschiedlichen Assertions, die wir zum Testen nutzen können. Die grundlegendsten sind:

`assertEquals`	- überprüft ob das erste und zweite Argument übereinstimmen
`assertTrue`	- überprüft ob das Argument eine wahre Aussage liefert.
`assertFalse`	- überprüft ob das Argument eine falsche Aussage liefert.

Assertions sind normale Methoden und können genauso behandelt werden. Sie können in deren Parametern weitere Methoden aufrufen, den Erwartungswert (das erste Argument) mithilfe einer Variable oder Methode zur Runtime bestimmen oder den Assert in einer Schleife aufrufen und mehrfach ausführen lassen.

Aufgabe

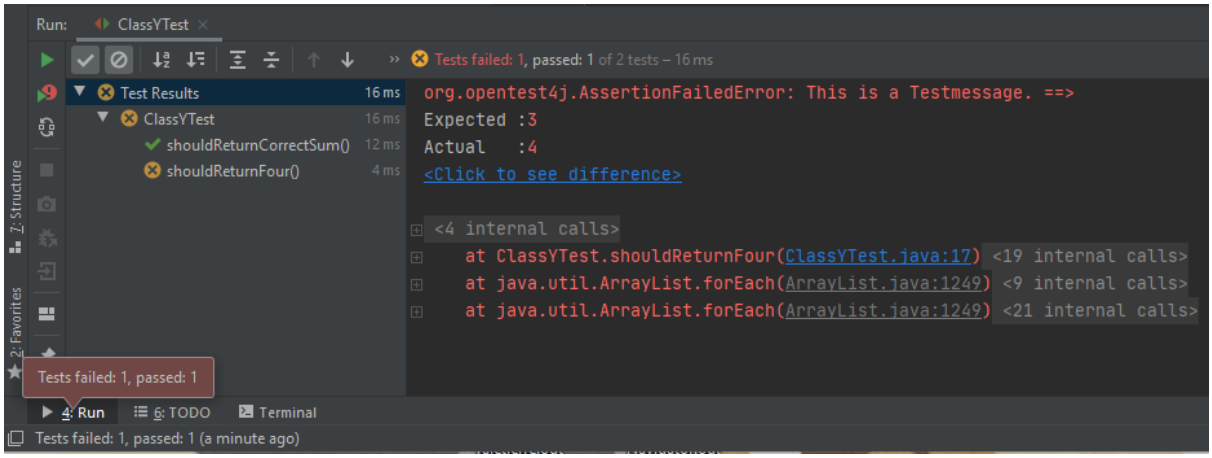
Schreiben Sie einen Test `shouldReturnCorrectSum`, der Ihrer Meinung nach die Anforderungen an eine Summen-Methode besser überprüft als unserer bisheriger Test. Nutzen Sie `assertEquals` und das was Sie bis jetzt über Java gelernt haben. Falls Ihnen noch weitere Tests für eine Summen-Methode einfallen, nutzen Sie es ruhig als Übung. Versuchen Sie danach, die Methode selbst zu schreiben und mithilfe Ihrer Tests die Funktionalität zu überprüfen.

Weitere Funktionen von JUnit

Mit der Annotation `@Disabled` können Sie einzelne Tests von der Ausführung ausschließen oder ganze Klassen wie zum Beispiel Ihre Testklasse. Wie bei allen Modulen von JUnit muss auch diese importiert werden, was identisch wie bei der `@Test` Annotation funktioniert. Alter-

nativ können Sie mit dem Import von `org.junit.jupiter.api.*` alle Module importieren. Dies kann unter Umständen aber Ihre Kompilierzeit erhöhen.

Bei Assertions ist der letzte Parameter ein optionaler String, der ausgegeben wird, falls die Assertion fehlschlägt. Damit können Sie zum Beispiel angeben, in welchem Schleifendurchlauf oder mit welchen Parametern die Assertion gestartet wurde etc…
Der String selbst kann natürlich auch wieder von einer Methode generiert werden, die im letzten Parameter angegeben wird.



So sieht ein normaler fehlgeschlagener Test auf der Konsole aus. Sie bekommen einen Stacktrace der fehlgeschlagenen Assertion sowie den Grund, warum sie fehlschlug. “This is a Testmessage.” ist der oben beschriebene String den wir `assertEquals` übergeben haben.

Der namensgebende Unit-Test

Die Gesamtfunktionalität einer Klasse kann oft über die `main`-Methode definiert werden. Ein einfacher Blackbox-Test würde also in dieser Methode testen. Aber nicht jede Klasse hat eine `main`-Methode, bzw. sind einige Klassen dafür da, anderen Funktionalität bereit zu stellen.

Größere Projekte umspannen dutzende Klassen und hunderte Methoden. Nur Tests für das Gesamtkonstrukt zu schreiben, ist umständlich und prohibitiv. Also würden wir in so einem Fall das Projekt in einzelne Teilstrukturen unterteilen.

Eine so genannte Unit ist ein semantisch zusammenhängender Teil unseres Codes. Units lassen sich auf mehreren Ebenen definieren. Wichtig ist hier, dass auch weiterhin das Blackbox-Prinzip gilt. Wir testen bei einer Unit nur den erwarteten Output für diese spezifische Unit.

Wie lassen sich Units definieren? Wenn wir eine Klasse schreiben, die arithmetische Funktionen bereitstellt, könnte die gesamte Klasse eine Unit sein. Für das Testen wäre es allerdings besser, wenn jede arithmetische Operation als Unit angesehen wird.

Units zu definieren hilft uns, die Grenzen unserer Methoden und Klassen zu sehen. Statt einer Methode für alle vier Operatoren können wir vier schreiben. Falls unsere Methoden in die

Bereiche von anderen Units eingreifen, sollten wir versuchen, sie zu vereinfachen oder aufzuteilen.
Falls mehrere unserer Units ähnliche Tests erfordern, zum Beispiel für eine Konvertierung von Gleitkommazahlen, ist es eine Überlegung Wert, die Funktionalität, die dort überprüft wird, in eine eigene Unit auszulagern. Also eine eigene Methode zu schreiben, die Kommazahlen konvertiert, und diese separat zu testen.

Wichtig beim Unit-Testing ist, dass jede Unit für sich genommen getestet wird. Wenn Code so eng miteinander verwoben ist, dass er nicht einzeln getestet werden kann sollte er als einzelne größere Unit betrachtet werden. Diese Betrachtungsweise hilft uns den Code semantisch zuzuordnen und zu kapseln.

Eine Übungsaufgabe

Abschließend wollen wir uns mit einem Feld befassen, in dem Test-Driven Design sehr prevalent ist. Wenn Sie in ein Fahrzeug steigen, haben Sie ein hohes Maß an Vertrauen, dass Ihnen nichts schlimmes passiert. Ein Grund dafür sind die rigorosen Tests, denen sich Automobilhersteller stellen müssen.

Wir wollen einen Blick auf einen Crashtest werfen und welchen Einfluss er auf eine `Auto`-Klasse hat.
In dem wir zum Beispiel einen Unit-Test `shouldDeployAirbagsWithin100ms()` schreiben setzen wir indirekt voraus, dass die `Auto`-Klasse Airbags besitzt und eine Methode `deployAirbags()`, die wir testen können.

Aufgabe

Schreiben Sie eine Klasse `Car` (oder `Auto`, wie Sie wollen), die von der bereits erstellten Klasse `Fahrzeug` erbt. Werfen Sie dann einen Blick auf diese Test-Klasse, die einen kleinen Crashtest simulieren soll. (/Java Crashkurs/Crashtest.java im Material-Ordner: https://gitlab.tubit.tu-berlin.de/algodat-sose2020/Material)

Erweitern sie die Klasse `Car` nun um die Felder und Methoden, auf die in dem Test zugegriffen wird. Sie müssen die Funktionalität nicht implementieren, können es aber gerne. Fallen ihnen noch weitere Dinge auf, die nicht in den Tests vorkommen, die in der `Car`-Klasse eventuell aber Intern benötigt werden?

Bedenken Sie auch, dass wir nicht direkt den Code einer `Car`-Klasse testen, sondern indirekt, ob sie über gewisse Attribute und Methoden verfügt und ob diese unseren Ansprüchen entsprechen.

Anhand dieses Beispiels möchten wir Ihnen zeigen, wie Tests dabei helfen können, Code zu schreiben, selbst wenn die Tests nicht einmal für den Code direkt geschrieben sind. Nehmen Sie sie also mehr als richtungsweisend wahr. Ihre Tests sollten natürlich später direkt auf Ihren Code zugeschnitten sein.

Recap

Test-Driven Design hilft uns, unserem Code Struktur zu geben, bevor wir ihn schreiben und gibt uns gleichzeitig die Mittel, ihn auf diese Struktur hingehend zu testen. Bevor Sie also anfangen zu programmieren, sollten Sie sich des Problems bewusstwerden, das Sie lösen wollen (oder sollen).

Falls Sie während des Programmierens merken, dass Sie nicht wissen, wie es weitergehen soll, treten Sie einen Schritt zurück und überprüfen Sie Ihre Tests. Ihre Tests sollten Ihnen stets den Rahmen vorgeben können.
Test-Driven Design ist ein Konzept, kein Gesetz. Sie müssen nicht jede Ihrer Codezeilen mit einem Test versehen oder jede Hilfsmethode vorher einplanen. Wichtig ist, dass Sie mit den Tests anfangen, da diese sonst oft zu kurz kommen. Wenn Sie später weitere Tests dazu schreiben umso besser!

Woche 3

In der letzten Woche dieses Crashkurses betrachten wir weitere Konzepte von Java, die für “Algorithmen und Datenstrukturen” hilfreich sind.

Der Java Generic Type

Eventuell haben Sie durch die Autovervollständigung der IDE oder eigene Experimente bereits gemerkt, dass in Java viele Standardmethoden mehrfach existieren. Die Unterschiede zwischen diesen sind oft nur der Rückgabewert bzw. einer der Eingabeparameter. So gibt es für eine Methode ein Dutzend verschiedene Varianten.

Was uns aber auffällt ist, dass die Varianten, die den Rückgabetyp ändern, nur für die unterschiedlichen Standardtypen existieren. Für die unendliche Anzahl möglicher komplexer Datentypen, mit denen Methoden interagieren können, gibt es in Java ein sehr mächtiges Konzept: Den Generic.

Der generische Datentyp ist ein Typ-Platzhalter, der zur Compile-Zeit verständlich macht, wo es sich zur Laufzeit später immer um denselben Datentyp handeln wird, ohne einzuschränken, was dieser Typ ist.

```
public class GenericDemonstration<T>{
    private T importantValue;

    public GenericDemonstration(T veryImportantValue){
        this.importantValue = veryImportantValue;
    }
}
```

Diese Klasse wurde mit einem Generic versehen. Die Namenskonvention in Java sieht einzelne große Buchstaben für diese vor, weshalb unser Generic mit `T` betitelt ist. Wie Sie sehen, steht `T` an Stelle unserer üblichen Typdeklaration.

Die spitzen Klammern rechts neben dem Klassennamen deklarieren den Generic für die Klasse. Dadurch kann er im Klassenrumpf eingesetzt werden. Wenn Sie ein Objekt dieser Klasse erstellen, müssen Sie dem Generic einen expliziten Typ zuweisen.

```
public static void main(String[] args) {
    /*
    Der Konstruktor erhält ebenfalls spitze Klammern,
    in welche der gewünschte Typ eingetragen wird.
    */
    GenericDemonstration<ClassY> demo = new
    GenericDemonstration<ClassY>(new ClassY());

    /*
    Der Compiler versteht, dass in der Zuweisung derselbe Typ
    gemeint ist. Er kann daher bei der Zuweisung weggelassen
```

```
    werden.
    */
    GenericDemonstration<ClassY> demo2 = new
    GenericDemonstration<>(new ClassY());
}
```

Eine Klasse, die Generics enthält, kann weiterhin normale Datentypen enthalten sowie Methoden, die gar nicht mit den Generics interagieren. Ebenso kann eine Klasse mehrere Generics benutzen. Diese werden dann von Kommas getrennt deklariert. Natürlich muss bei einer Instanzierung der Klasse für jeden Generic ein Datentyp angegeben werden.

```
public class GenericDemonstration<T,E,D>{
    private T importantValue;
    private int unimportantNumber;
    private String blandString;

    public GenericDemonstration(T veryImportantValue, int meh,
    String hereBecauseIHaveToBe){
        this.importantValue = veryImportantValue;
        this.unimportantNumber = 9001*meh;
        this.blandString = hereBecauseIHaveToBe;
    }
    public GenericDemonstration(){

    }

    public GenericDemonstration(T veryImportantValue){…}

    public static void main(String[] args){
        /*…*/

        GenericDemonstration<ClassY, ClassYTest, ClassYTests>
        demo3 = new GenericDemonstration<>();
    }
}
```

Der Generic ist also ein effizienter Weg, einer Klasse die Fähigkeit zu geben, mit verschiedenen Klassen reibungslos interagieren zu können. Allerdings können Sie nicht ohne weiteres Methoden auf dem Generic aufrufen. Die Generics erben direkt von der `Object`-Klasse und haben daher unter anderem nur die allgemeinen Methoden `toString()`, `equals()` und `clone()`. Sie eignen sich aber perfekt dafür, Datenstrukturen allgemein kompatibel zu machen.

Aufgabe

Probieren Sie es aus. Schreiben Sie eine Klasse, die einen Generic enthält, und eine Methode, die diesen Generic übergeben bekommt. Die Methode soll den Generic in einen String einbetten und auf der Konsole ausgeben. Testen Sie es mit den unterschiedlichen Fahrzeugklassen, die Sie in den vorherigen Lektionen erstellt haben. Wie gehen Sie mit dem Problem um, dass ein Generic nur einen expliziten Datentyp haben kann?

Von Stapeln und Schlangen – Eine kurze Wiederholung

Höchstwahrscheinlich haben Sie im Studium, der Schule oder in Ihrer Freizeit schon einmal eine Queue oder einen Stack implementiert. Trotzdem geben wir Ihnen hier eine kurze Auffrischung.

Eine Queue (Warteschlange) ist eine einfache Listen-artige Datenstruktur, bei der Elemente am Ende angefügt und vom Anfang aus abgearbeitet werden. Es gilt FirstInFirstOut, das Element, das zuerst hinzugefügt wurde, wird auch zuerst entnommen. Ein Stack (Stapel) wiederum arbeitet auch stets mit dem ersten Element wie eine Queue, fügt Elemente allerdings ebenfalls am Anfang ein. Es gilt LastInFistOut, das Element, das zuletzt hinzugefügt wurde, wird als erstes entnommen.

Diese zwei Datenstrukturen sowie die Liste, auf der sie basieren, sind elementare Bauelemente in der Informatik und der Programmierung. Sie müssen sich daher nicht die Mühe machen, diese immer wieder zu implementieren.

Wie die meisten gängigen Programmiersprachen besitzt auch Java eine Fülle von Standardklassen, die uns während der Arbeit zur Verfügung stehen. Die Klassen `String` und `Array` zum Beispiel sind Teil der Standardbibliothek von Java. Das Package `java.util`. enthält eine Sammlung von Klassen, die Sie verwenden können und sollten, so zum Beispiel `Stack` und `Queue`. Das Package `java.io` wiederum gibt Ihnen alles was Sie brauchen, um mit Dateien und externen Inputs zu arbeiten.

Wie finden Sie heraus, welche Klassen und Features bereits für Sie implementiert sind? Diese offiziellen Klassen haben eine rigorose Online-Dokumentation im so genannten Javadoc-Format. Wenn Sie in der Suchmaschine Ihres Vertrauens zB. “Java 14 API Stack” eingeben, sollte das erste Ergebnis diese Dokumentation sein. Oft gefolgt von der Dokumentation einer früheren Java Version. Die Webseite der Dokumentation ist
Docs.oracle.com, mithilfe einer Suchmaschine finden Sie allerdings meist einfacher zum Ziel.

Aufgabe

Wenn Sie die Dokumentation öffnen, sehen Sie eine Erklärung, wozu die Klasse fähig ist, ein Inhaltsverzeichnis mit allen Attributen, Konstruktoren und Methoden sowie Informationen über die Erbbaumhierarchie und eventuelle Interfaces.

Machen Sie sich mit dem Layout vertraut. Suchen Sie dann nach der Dokumentation der Java `Queue`.

Ändern Sie den Test aus der vorherigen Aufgabe dahingehend, dass alle zu testenden Objekte in eine Queue eingefügt werden und nacheinander abgearbeitet werden. Wie Sie bereits in der Doc feststellen können, nutzen Stacks und Queues in Java ebenfalls Generics.

Collections

Stacks und Queues implementieren beide das `List` Interface. Das List Interface ist allerdings nur eine Erweiterung des `Collections` Interface. Dieses Interface gibt Ihnen eine Vielzahl an Methoden vor, die eine Klasse dieser Art haben sollte. Suchen Sie einmal nach dem Collections Interface und lesen Sie die ersten Paragraphen (Sie müssen nicht den ganzen Text lesen).

Wenn Sie also eine Datenstruktur implementieren, die annähernd einer Sammlung von Daten entspricht, empfiehlt es sich einen Blick auf dieses Interface zu werfen. Alternativ schauen Sie sich die Liste an Klassen an, die dieses Interface implementieren, um Datenstrukturen zu finden, die Sie für Ihre Aufgabe benutzen können. Wie zum Beispiel Queues, Stacks oder Linked Lists.

Das Collections Interface repräsentiert allerdings auch einen Kerngedanken von Java, nämlich den der Interoperabilität. Wenn Sie eine Klasse mit diesem Interface versehen, dann können andere Klassen einfacher mit Ihrer interagieren. Wenn sich alle Programmierer einigen, auf Basis zentralisierter Interfaces zu programmieren, führt das zu einem hohen Grad von Kompatibilität.

Die Javadoc

Diese Standardklassen-Dokumentationen sind alle mithilfe von Javadoc Kommentaren verfasst, die automatisch Code-Kommentare in eine HTML-Seite mit diesem Format konvertieren. Die Javadoc ist ein allgemein anerkanntes Format und hilft dabei, auf die notwendigen Informationen aufmerksam zu machen, die ein Kommentar enthalten soll. Außerdem führt die Standardisierung von Kommentaren zu einer einheitlichen und wiedererkennbaren Form.

Damit Sie diese Funktion nutzen können, müssen Sie einigen Style-Richtlinien folgen. Javadoc-Kommentare stehen vor Methoden, Attributen, Konstruktoren oder Klassen. Jeder Blockkommentar wird mit `/**` eröffnet und mit `*/` wieder geschlossen.

Als allererstes sollte eine kurze Schilderung stehen, was der Code bezweckt. Diese taucht in der Zusammenfassung am Anfang der Dokumentation auf, in der alle Methoden, Attribute etc… aufgelistet werden. Oder im Falle der Klasse selbst ist es der Einführungstext der Seite.

Folgend auf den Text kommen so genannte “Block Tags”. Sie dienen dazu die Dokumentation zu verknüpfen und sind Schlagwörter für den Doc-Processor. Sie sind folgendermaßen aufgebaut.

```
@return     this method always returns null
```

Eine Methode sollte zum Beispiel einen `@return` Tag haben wenn sie einen Rückgabewert hat. Mit diesem Tag kann man den Rückgabewert mit einem Kommentar versehen. Der zweite wichtige Tag ist `@param`.

```
@param  myParameter this parameter is only for show
```

Jeder Übergabeparameter einer Funktion bekommt seinen eigenen `@param`-Tag, in dem der Variablenname des Parameters angegeben wird, gefolgt von einer kurzen Erklärung, was er bezweckt. Der Datentyp des Parameters wird dann automatisch erfasst und falls existent in der Javadoc verlinkt.

```
/**
 * This class is used to generate and house all Tests for ClassY
 */
public class ClassYTests{
    /**
     * Container for our generated tests.
     */
    private LinkedList<ClassYTest> listOfTests;
    /**
     * Creates a new container for ClassY tests. Starts out
     * without any tests generated.
     */
    public ClassYTests(){
        this.listOfTests = new LinkedList<>();
    }
/**
 * This method will generate any number of Tests requested.
 * @param numberOfRequests the number fo tests that will be
 * created
 */
    private void generateTest(int numberOfRequests){
        for(int i = 0; i < numberOfRequests; i++){
            this.listOfTests.add(new ClassYTest());
        }
    }
}
```

Um diese Kommentare im gerenderten Format zu sehen können Sie sie von IDEA generieren lassen. Wie Sie in IDEA mit Javadocs arbeiten können Sie hier nachlesen:
https://www.jetbrains.com/help/idea/working-with-code-documentation.html

Exceptions

Beim Programmieren, aber auch bei der Verwendung von Programmen treten unvermeidlich Fehler auf. In Java wird dann je nach Fehler eine entsprechende Exception “geworfen”. So z.B. eine `ArrayIndexOutOfBoundsException`, wenn Sie versuchen, eine Stelle in einem Array aufzurufen, die über das Array hinausgeht.
Java bietet aber auch die Möglichkeit, selbst Exceptions zu werfen:

```
if(i >= 10){
    throw new ArrayIndexOutOfBoundsException("Index zu hoch!");
}
```

Es können aber auch von Java geworfene Exceptions abgefangen werden. Dazu dienen try/catch-Blöcke:

```
try{
    int a = array[i];
}catch(ArrayIndexOutOfBoundsException e){
    System.out.println("Der Index ist zu hoch!");
}
```

Hierbei wird ein Stück Code, das einen Fehler verursachen könnte, in den `try`-Block geschrieben, während im `catch`-Block die Exception, die auftreten könnte, abgefangen wird. Wird die vermutete Exception von Java geworfen, wird der Code im `catch`-Block ausgeführt, wird sie nicht geworfen, wird der `catch`-Block ähnlich wie bei einer `if`-Abfrage einfach übersprungen.
Wenn Sie sich nicht sicher sind, was für ein Fehler in Ihrem Code auftreten könnte, können Sie als Argument des `catch`-Blocks auch einfach den Typ `Exception` schreiben. Alle genauer spezifizierten Exceptions erben von dieser `Exception`-Klasse, deshalb kann so eine beliebige Exception abgefangen werden.

Schlusswort

Java ist eine weitverbreitete und vielseitige Sprache. Die starke Community und die JavaVirtualMachine machen sie zu einer exzellenten Wahl für Anwendungsentwicklung und ideal für den Einstieg in Objektorientierte Programmierung.

Wir hoffen, Sie konnten mit Hilfe dieses Kurses ein Grundverständnis von Java entwickeln oder Ihre Kenntnisse auffrischen und fühlen sich jetzt gewappnet, tiefer in Java einzusteigen und die Hausaufgaben für “Algorithmen und Datenstrukturen” zu bearbeiten.

Falls Sie an weiterer Übung interessiert sind, bieten Jetbrains, die Ersteller von IntelliJ IDEA, einen ausführlichen kostenfreien Javakurs vom Anfänger-Level bis zur knackigen Herausforderung.

Über die Plattform https://hyperskill.org/ können Sie auf alle Module dieses Angebots zugreifen. Wir empfehlen das Modul "Coffee Machine" oder für eine etwas größere Herausforderung "Smart Calculator". Einiges, was Sie hier gelernt haben, wird in diesen Kursen wiederholt, Sie gehen aber darüber hinaus und bieten die Möglichkeit, an einem abgeschlossenen Projekt zu arbeiten.

Außerdem können Sie die Hyperskill-Lektionen direkt in IDEA bearbeiten, wenn Sie die IDEA-Educational Version installieren. Wie Sie Zugriff auf die Lektionen erhalten. lesen Sie hier: https://www.jetbrains.com/help/idea/product-educational-tools.html