Programmiermethodiken

Lehrveranstaltung

Vorlesung

Dozent: Zangerle Eva (???)

Termin: **Mittwoch 14:15 – 15:00, Hörsaal F**
Donnerstag 08:15 – 10:00, Hörsaal F

Klausur-Termine: **28.06.2017, 13:0 – 16:00, HSB3**
Anmeldung ab: **24.05.2017**

Links:

- https://lfuonline.uibk.ac.at/public/lfuonline_lv.details?lvnr_id_in=703016&sem_id_in=17S
- https://dbis.uibk.ac.at/teaching

Literatur:

- Hanspeter Mössenböck, Sprechen Sie Java? Eine Einführung in das systematische Programmieren, dpunkt.verlag, 5. Auflage, 2014
- Reinhard Schiedermeier, Programmieren mit Java, Pearson Studium, 2. Auflage, 2010
- Bernhard Lahres, Gregor Rayman, Objektorientierte Programmierung. Das umfassende Handbuch, Galileo Computing, 2. Auflage, 2009, online verfügbar
- David J. Barnes, Michael Kölling, Java lernen mit BlueJ. Eine Einführung in die objektorientierte Programmierung, Pearson Studium, 5. Auflage, 2013
- Bernhard Steppen, Einstieg in Java 7, Galileo Computing, 4. Auflage, 2012
- Christian Ullenboom, Java ist auch eine Insel: Einführung, Ausbildung, Praxis, Galileo Computing, 11. Auflage, 2014, (10. Auflage ist online verfügbar)

Proseminar

Dozent: Silbernagl Doris

Termin: **Donnerstag 10:15 – 12:00, Hörsaal 10**

Prüfungen: **2 Tests**
Termine: **02.05.2017**
06.06.2017

Anzahl erlaubter, unentschuldigter Fehlstunden: **2**

Links:

- https://lms.uibk.ac.at/auth/RepositoryEntry/4147020284/CourseNode/95281602340753
- https://dbis.uibk.ac.at/teaching

Kompendium

Inhaltsverzeichnis

Allgemeines

Klassifizierung von Programmiersprachen

Programmierparadigma

Damit meint man die Sichtweise auf und Umgang mit den zu verarbeitenden Daten und Operationen.

- Imperatives Programmierparadigma
  Dabei handelt es sich um die streng sequenzielle Abarbeitung von Anweisungen.
- Objektorientiertes Programmierparadigma
  Verfügt über weitergehende Konzepte wie Klassen, Vererbung und Polymorphie
- Funktionales Programmierparadigma
  Diese kennt keine Wertzuweisung, sondern nur eine Reihe von Funktionsaufrufen, die eine Eingabe in einen Ausgabe transformiert.
- Logisches Programmierparadigma
  Verwendet Logik zur Darstellung und Lösung von Problemen. Dabei besteht das Programm aus einer Menge von Axiomen, aus denen der Interpreter eine Lösungsaussage berechnet.

Beispiel: Konkatenation zweier Listen

Imperativ:

1. Aktuelles Element E = Kopf von A
2. Solange Nachfolger von A existiert, erweitere E um Nachfolger von A
3. Setze Nachfolger von E auf Kopf von B, erweitere wieder bis Ende

Objektorientiert:

1. Interner Aufbau wie oben
2. Aufruf erfolgt aber über ein Objekt und verändert dieses: A.append(B);

Funktional:

1. append([], B) = B
2. append([X | A], B) = [X | append(A,B)]
3. Aufruf “append([1,2], [4,5]) liefert [1, 2, 3, 4]

Logisch:

1. append(A, B, C) ist wahr, gdw. C Konkatenation von A und B ist
2. append([], B, B)
3. append([X | A], B, [X | C]) ← append(A, B, C)
4. Aufruf append([1, 2], [4, 5], Xs) liefert Antwort Xs = [1, 2, 4, 5]
5. Aufruf append(A, B, [1, 2, 3]) liefert Antworten A = [], B = [1, 2, 3] oder A = [1], B = [2, 3] oder A = [1, 2], B = [3] oder A = [1, 2, 3], B = []

Abstraktionsgrad

Der Abstraktionsgrad definiert, wie weit sich die Semantik der einzelnen Sprachkonstrukte von den Grundbefehlen des Mikroprozessors unterscheidet.

Maschinensprache

Wird direkt vom Prozessor verstanden

Assembler

Einsatz symbolischer Namen, Sprungmarken, etc.

Höhere Programmiersprache

Kontrollstrukturen, Datentypen, etc.

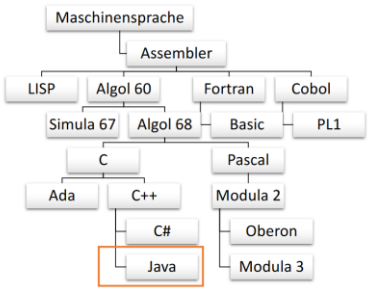
Objektorientierte Programmiersprachen

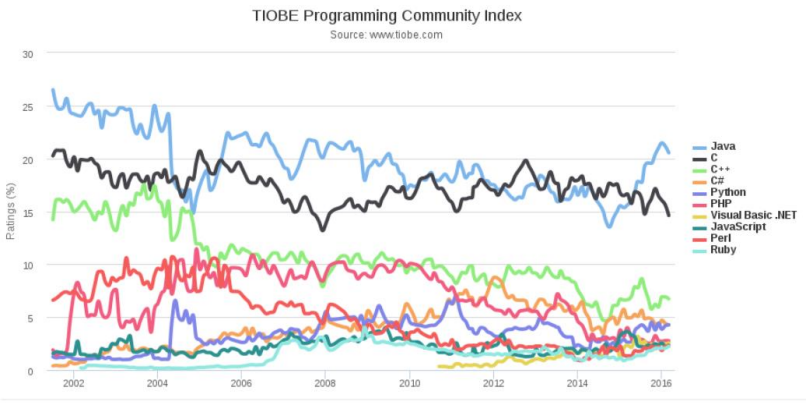
Daten und Methoden bilden eine Einheit

Deklarative Programmiersprache (Funktional/Logisch)

Kein Unterschied zwischen Daten und Operationen („was“ wird geschrieben)

Abstammung





Ausführungsschema

Übersetzende Programmiersprachen

Quelltext eines Programms wird vor der Ausführung durch einen Übersetzer in eine Zielsprache übersetzt, wodurch eine hohe Ausführungsgeschwindigkeit garantiert wird.

Interpretierende Programmiersprachen

Quelltext wird zur Laufzeit von einem Interpreter eingelesen und Befehl für Befehl abgearbeitet, was natürlich langsamer ist, dafür aber flexibler ist.

Mischformen, wie Java

Programm wird in einen Zwischencode übersetzt, welcher von einer virtuellen Maschine interpretiert wird.

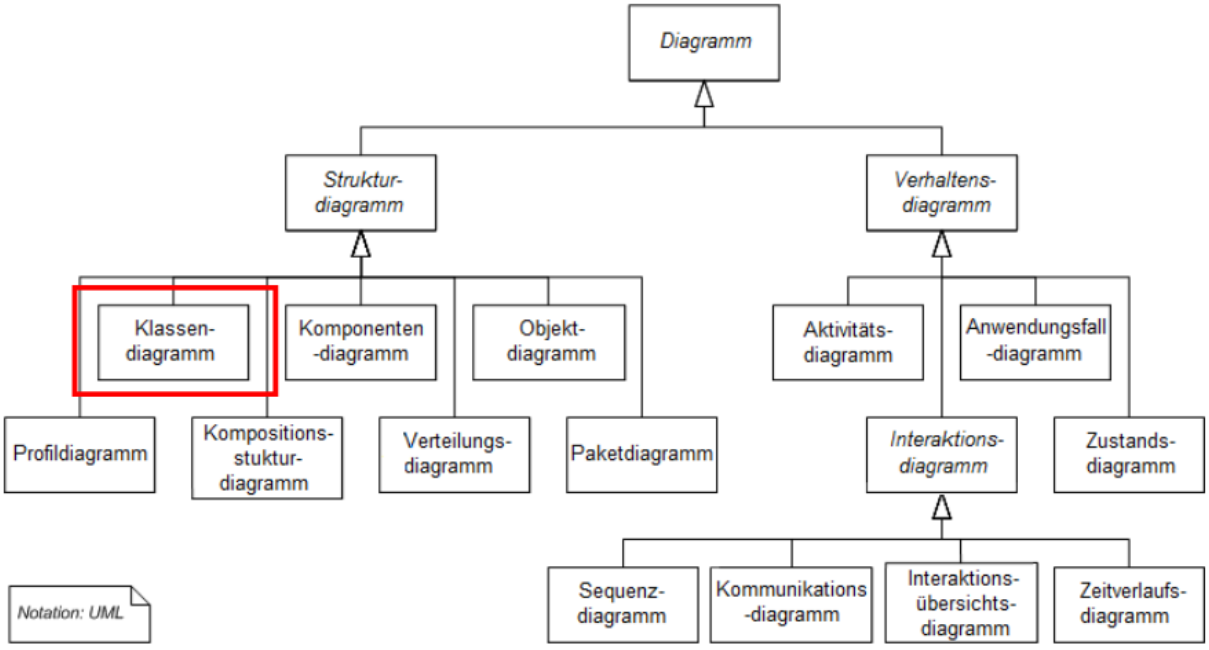
UML

Unified Modeling Language

Sie ist eine standardisierte, ausdrucksstarke Modelleriungssprache, mit deren Hilfe Softwaresysteme besser **entworfen, analysiert** und **dokumentiert** werden.

Während Unified für eine Unterstützung während des gesamten Entwicklungsprozesses und eine Unabhängigkeit von Entwicklungswerkzeugen/Sprachen/Anwendungsgebieten steht, ist UML kein vollständiger Ersatz für Textbeschreibungen oder gar ein Vorgehensmodell!

Obwohl es etliche unterschiedliche Diagrammarten in der UML gibt, beziehen wir uns im folgenden lediglich auf das Klassendiagramm. Für das Sequenzdiagramm wird auf die Vorlesung über Entwurfsmuster verwiesen.



Grafik übernommen von http://de.wikipedia.org/wiki/Unified_Modeling_Language

Sprachkonzept	Notation
Klasse	Name
Klasse mit Abschnitten	Name Attribut1 Attribut2 Operation1 Operation2

Notation für Klassen

Attribute

Für die Sichtbarkeit von Attributen und Operationen verwenden wir

- \+ = public
- # = protected
- \- = private
- ~ = default

Klassenattribute werden unterstrichen, Instanzattribute nicht.
Außerdem dürfen wir zusätzliche Eigenschaften angeben, wie
<<readOnly>>, <<ordered>>, <<unique>>, <<redefines <Operationname>>>

Multiplizität

Wir können auch spezifizieren, wieviele Werte ein Attribut aufnehmen kann

```
Attribut:Typ[a..b]
```

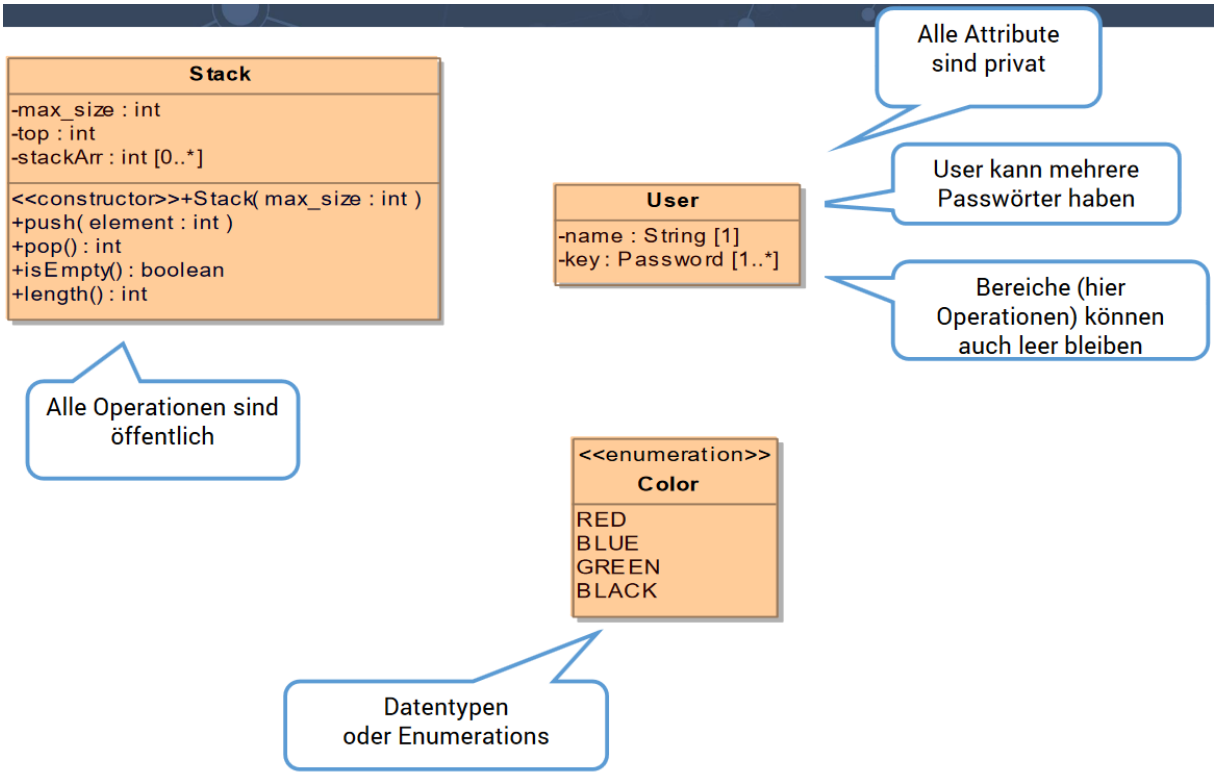
Wobei mindestens a .. höchstens b, a und b natürliche Zahlen sind. * statt b bedeutet beliebig viele.

Operationen

Obligate Angabe einer Parameterliste für jede Operation (auch wenn diese leer ist). Für jeden Parameter muss midnestens der Name und sein Typ (sowie Multiplizität) angegeben werden. Rückgabetyp ist natürlich auch erwünscht. Klassenoperationen können wir markieren, indem wir sie unterstreichen. Die Sichtbarkeit entspricht der, der Attribute.

Zusätzliche Eigenschaften wären:
<<abstract>> (oder kursiv geschrieben), <<leaf>> (wenn final), <<raisedException>> (wie throws), <<enumeration>>, <<constructor>>

Beispiele



Assoziation

Die Grundform ist eine binäre Assoziation, die zusätzlich benannt werden kann. Auch die Rollen der beiden binären Operanden, deren Multiplizität, Eigenschaften und eine mögliche Naviagationsangabe (Richtung durch Pfeil, Keine Navigation durch X) sind erlaubt.

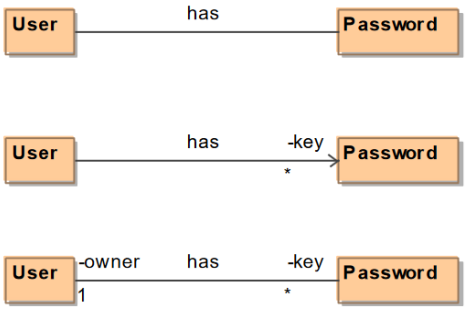
Aggregation

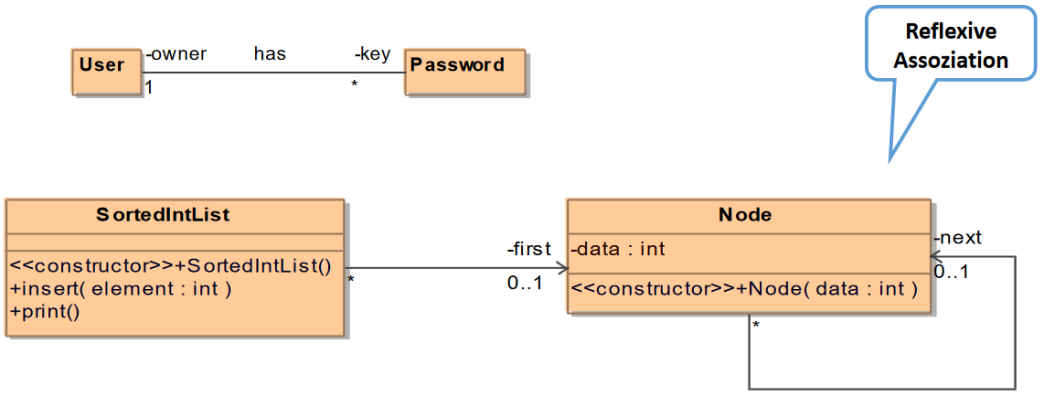
Ist eine spezielle Assoziation (Teile-Ganzes-Beziehung).

Eine Komposition aäre eine speziellere (strengere) Form der Aggregation mit folgenden Einschränkungen:

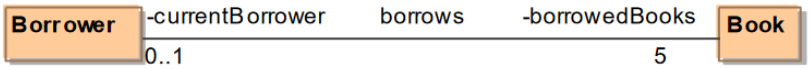
- ❖ Ein Teil darf Kompositionsteil höchstens eines Ganzen sein
- ❖ Multiplizität von 1 bedeutet, dass das Teil nur solange existiert, wie sein Ganzes.

Beispiele





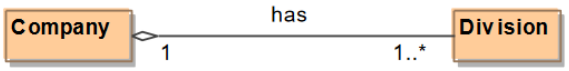
Hier ein Beispiel für eine Java-Implementierung:



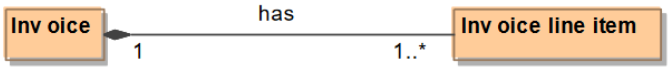

```
public class Borrower{
     private Book[] borrowedBooks;
     private int numBooks;
     ...
     public Borrower(){
          numBooks = 0;
          borrowedBooks = new Book[5];
     }

     public void borrowBook(Book b){
          ...
          borrowedBooks[numBooks] = b;
          numBooks++;
          b.setBorrower( this );
          ...
     }
}
```

```
public class Book{
     private Borrower currentBorrower;
     ...
     public void setBorrower(Borrower bw){
          currentBorrower = bw;
     }
}
```

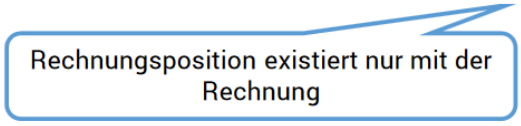
Beispiel für die Aggregation/Komposition



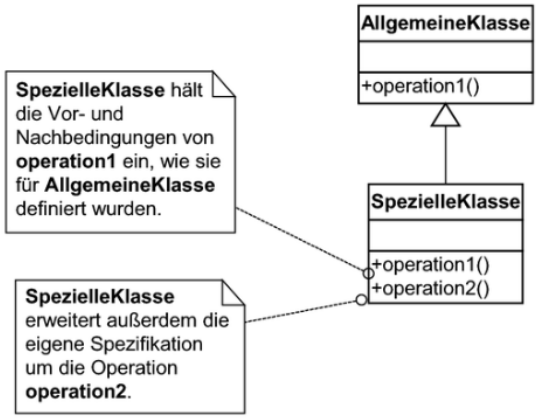
Aggregation

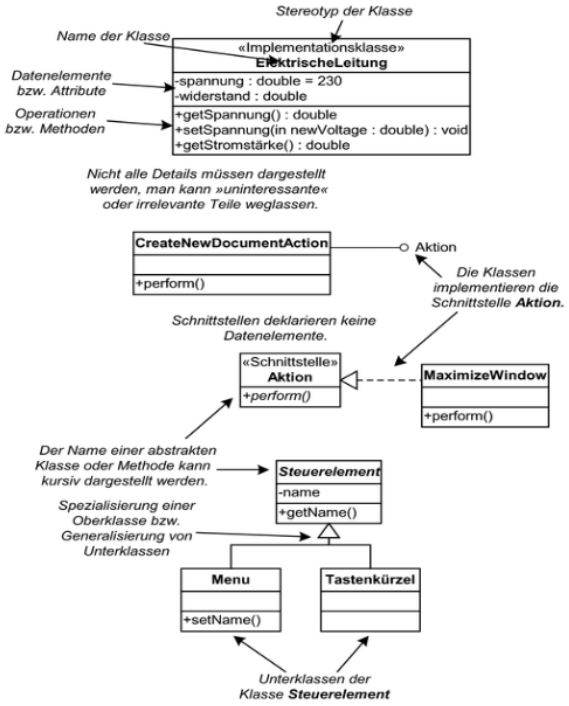


Komposition



Vererbung, Schnittstellen





Tests

Testen ist ein Vorgang, um Fehler zu finden. Es dient also zur Sicherstellung von Softwarequalität. Dazu vergleichen wir spezifizierte Anforderungen mit gelieferten Ergebnissen.

Grundsätzlich gibt es zwei Vorgehensweißen:

Black-Box-Prinzip

Dabei werden nur die Anforderungen des Programms berücksichtigt, das heißt eine Schnittstelle gibt die Testfälle vor. Der Source-Code spielt keine Rolle (Programm wird als Black-Box betrachtet). Folglich erfordern auch Änderungen am Quelltext keine Änderung der Implementierung der Tests.

White-Box

Test orientiert sich am Quelltext des Programms. Damit möchte man alle Codeabschnitte erfassen und die einzelnen Zweige, Schleifen etc. Explizit

austesten („Code Coverage“). Natürlich erfordern Änderungen am Quelltext auch eine teilweise neu-implementierung von Tests.

Es sind immer drei Teilnehmer im Test-Prozess:

- Entwickler
  - Unit-Tests (Methode, Klasse, Projekt)
  - Komponententest (Klasse, Package)
  - Integrationstest (Teilsystem)
- Tester
  - Funktionstest (Teilsystem, Gesamtsystem)
  - Applikationstest (Gesamtsystem, Applikation)
  - Usability-Test (Applikation)
- Kunde
  - Abnahmetest (Applikation)

Abgrenzung zur Verifikation

Die Verifikation ist ein formaler Korrektheitsbeweis, bei dem mithilfe von formalen und mathematischen Methoden nachgewiesen wird, dass ein Programm richtige Ergebnisse produzieren kann. Sie ist eine endgültige Aussage zur Korrektheit und schon für kleinste Programme ein sehr aufwändiges Unterfangen.

Ein Test hingegen ist ein systematisches Ausprobieren. Dabei kann aber nur die Anwesenheit von Fehlern nachgewiesen werden, nicht aber deren Abwesenheit. Grundsätzlich braucht man sehr viele Testfälle, um ein Programm ausführlich zu testen. Das ist dann aber immer noch keine endgültige Aussage!

Unit-Tests

Ein Unit-Test dient zum Testen einer sehr kleinen Einheit des Programms (Unit). Die Unit wird isoliert vom Rest des Gesamtsystems betrachtet, man will also nur die Funktionsweiße einer einzelnen Einheit testen – ohne Einfluss des Gesamten.

Dadurch können sie automatisiert werden und teilweise direkt als Programm implementiert werden. Teilweise werden diese Unit-Tests auch als Programmiermethodik verwendet: **Test-Driven Development.**

Test-Driven Development

Dabei werden zuerst Ergebnisse in Form von Testfällen implementiert. Danach wird erst der Code entwickelt, wobei nur leere Methoden erzeugt werden → Übersetzung funktioniert, Tests schlagen fehl.
Dann wird schrittweise jede Methode implementiert, und mit den Tests geprüft – solange, bis alle Tests durchlaufen.

Dies ist natürlich ein sehr professioneller Ansatz, Tests werden zuerst erstellt, können also nicht verdrängt werden, Funktionaltität wird schon vorgegeben und das gesamte System muss bereits in sich selbst durchdacht werden. Allerdings ist es ein sehr aufwändiges Unterfangen.

TestNG

Bietet ein einheitliches Framework zur Organisiation und systematischen Durchführung von Unit-Tests. NG = New Generation. Sie ist nicht Teil des JDK,

muss unter testng.org heruntergeladen werden und in die CLASSPATH das entsprechende jar-Archiv eingebunden werden.

TestNG wird für Unit-Tests verwendet, wobei die Tests für isolierte Klassen/Methoden entweder gelingen (grün) oder fehlschlagen (rot). Im letzteren Fall wird zwischen Error und Failure unterschieden. Failures werden erwartet, Errors treten unerwartet auf.

Einzelne Tests sind Java-Objekte, Testmethoden müssen mit @Test versehen werden. Sie vergleichen stets einen vom Programmierer festgelegten Wert mit dem berechneten Wert der Methode. Dafür werden asserts verwendet *(siehe 1 Semester „Einführung in die Programmierung“)*

Die Signatur der Test-Methoden sind vorgegeben, nur der Name darf frei gewählt werden. Beispiel:

```
@Test public void test() {
    int want = …;
    int have = …;
    assertEquals(want, have);
}
```

Wobei hier angemerkt sei, dass assertEquals eine Definition von Equals benötigt.

TestNG bietet viele Assert-Methoden an. Wenn du diese sehen willst, dann schau in die Doku.
Allerdings sei hier noch verraten, dass alle Vergleichsmethoden mit zusätzlichem String-Parameter an erster Stelle überladen sind, mit dem eine Message für den Fall eines Fehlschlags mitgegeben werden kann.

Test-Klassen

Für jede zu testende Klasse gibt es eine Test-Klasse. Sie importiert benötigte TestNG und Methoden, und definiert Testmethoden und Verwaltungsmethoden.

Zweck der Methoden wird mit Annotations festgelegt

`@Test`
- Testmethode (mehrfach)

`@BeforeMethod`
- Wird **vor** jedem einzelnen Test aufgerufen.

`@AfterMethod`
- Wird **nach** jedem einzelnen Test aufgerufen.

`@BeforeClass`
- Wird einmal vor allen Tests aufgerufen.

`@AfterClass`
- Wird einmal nach allen Tests aufgerufen.

… viele mehr, siehe Dokumentation.

Erwartete Fehler

Um unzulässige Eingaben zu testen (also dass z.B.: die Exceptions geworfen werden) können wir die erwartete Exception vorgeben. Dann scheitert der Test, wenn die Exception nicht geworfen wird.

Junit 4

Ist ähnlich wie TestNG ein Unit-Test-Modul. TestNG ist laut den Vorlesungsfolien besser.

Objektorientierung

Eine gute Software muss vorallem **Korrektheit, Effizienz, Benutzerfreundlichkeit** und **Wartbarkeit** vorweisen können. Dabei ist **Komplexität der größte Gegenspieler**.

Diese Komplexität soll mittels Modularität in den Griff bekommen werden, das heißt, wir zerstückeln das Programm in kleine Häppchen, die in sich geschlossen arbeiten können. Genau dafür wurde Objektorientierung erfunden – auch wen diese nicht automatisch zu besserer Software führt, und auch nicht alle Prinzipien der Modularität unterstützen, ist Objektorientierung dennoch ein sehr gutes Hilfsmittel.

Objekte

Alles ist ein Objekt, Objekte haben eine Identität, einen Zustand und ein Verhalten.

Objekte kommunizieren durch Datenaustausch, Objekte haben ihren eigenen Speicher und jedes Objekt ist ein Exemplar einer Klasse.

Diese Klasse modelliert das gemeinsame Verhalten ihrer Objekte.

Einführung in die Objektorientierung

Hier verweise ich auf den Foliensatz PM-3-Objektorientierung.pdf,
weil mir des zu hart war de dinge de ich eh schon kann zusammenzufassen.
Hier stehen also nur die wichtigsten Infos für mich selber.

Grundelemente der OO

Datenkapselung / Information Hiding

Ziel ist es hierbei, Eigenschaften eines Objekts nur über Methoden zur Verfügung zu stellen. Das heißt, die Daten eines Objekts werden gekapselt. Die Implementierung der Methoden ist ebenfalls verborgen.

Der Aufrufer bekommt also wirklich nur die Information, die er braucht – nämlich die vom aufrufenden Objekt berechneten Daten. Dafür ist die Schnittstelle eines Objekts da -> Sie ist die Menge der Operationen eines Objekts.

Vererbung

Eh klar.

Polymorphie

Polymorphie = Vielgestaltigkeit. Das heißt, eine Variable kann, abhängig von ihrer Verwendung, unterschiedliche Datentypen annehmen. Damit wird flexiblere Software ermöglicht.

Konstruktoren

Wichtig für Java ist zu wissen, dass Objektvariablen & Klassenvariablen automatisch initialisiert werden, sollten sie nicht explizit über den Konstruktor mit einem anderen Wert initialisiert werden:

Datentyp	Initialisierung
int	0
double	0.0
boolean	false
char	\u0000
String	null
Referenztypen	null

Java erzeugt automatisch einen Default-Konstruktor, wenn keiner angegeben wird. Sobald aber einer existiert, müsste der Default-Konstruktor manuell hinzugefügt werden.

Typsystem

Ein Typsystem ist Bestandteil einer Programmiersprache und dient zur Überprüfung, ob die Verwendung (von z.b. einer Operation auf ein Objekt) mit den Regeln des Typsystems verträglich ist.

Statisches Typsystem

Überprüfung zur Übersetzungszeit. (Java, C++, C#)

Typ von Variablen und Parametern wird im Quelltext deklariert, schränkt ein, welche Objekte einer Variable zugewiesen werden können. Diese unpassende Zuweisung erkennt schon der Compiler.

Daraus resultiert natürlich, dass der Compiler bessere Optimierungen durchführen kann, die Programmstruktur übersichtlicher, die Entwicklungsumgebung besser unterstützt und Fehler frühzeitig erkannt werden.

Dynamisches Typsystem

Überprüfung zur Laufzeit. (PHP, Python)

Variablen sind keinem Typ zugeordnet, können beliebige Objekte referenzieren und die Überprüfung, ob eine Operation auf einem Objekt erlaubt ist, wird zur Laufzeit überprüft.

Vorteile sind folglich natürlich die Flexibilität und keine expliziten Typumwandlungen. Dafür ist der Code unübersichtlicher, Klassen und Typen sind entkoppelt.

Stark typisiert

Eine stark typisierte Programmiersprache stellt sicher, dass Variablen immer auf Objekte verweisen, die auch die Spezifikation des Typs erfüllen, der für die Variablen deklariert ist.

Schwach typisiert

Programmiersprachen, welche schwach typisiert sind, erlauben die Zuweisung eines Objekts zu einer Variable, dessen Spezifikationen nicht zwangsläufig erfüllt werden.

Typsystem und Typisierung sind unabhängig voneinander!

Getter-Setter

Joah, was halt getter und setter so tun.

Intensiver Gebrauch ist ein Zeichen von schlechtem Code.

Zugriffsattribute

- public
  Zugriff aus beliebiger Klasse erlaubt
- private
  Zugriff nur aus Methoden derselben Klasse
- protected
  Zugriff aus beliebiger Klasse desselben Pakets und aus Unterklassen
- default (kein Attribut)
  Zugriff aus beliebiger Klasse desselben Pakets

Wobei auch hier wiederrum zwischen **Klassenbasierte Sichtbarkeit** und **Objektbasierte Sichtbarkeit** unterschieden werden muss. Der Unterschied liegt hauptsächlich darin, dass bei private in Klassenbasierter Sichtbarkeit ein Objekt auf die privaten Daten eines anderen Objekts derselben Klasse zugreifen darf.

Dies (klassenbasiert) wird in Java unterstützt. Zum Beispiel:

```
public Rectangle(final Rectangle b) {
  this.width = b.width; // Kopier - Version  → private
  this.length = b.length;
}
```

Flache oder Tiefe Kopie

Flache Kopie (shallow copy)

Wird nur mit Wertzuweisung kopiert, spricht man von einer flachen Kopie. Das heißt es wird nur das Objekt (nicht aber die darin enthaltenen Objekte) kopiert.

Tiefe Kopie (deep copy)

Es wird auch jedes Objekt kopiert, rekursiv, bis es nichts mehr zum Kopieren gibt. Damit enthalten Original und Kopie die gleichen Daten, können aber getrennt verwendet werden.

Kohäsion und Kopplung

Kohäsion

Sie beschreibt wie gut eine Methode tatsächlich genau eine Aufgabe erfüllt oder wie genau abgegerenzt die Funktionalität einer Klasse ist. Eine hohe Kohäsion deutet auf eine gute Trennung der Zuständigkeiten hin. Sie hilft also der Wiederverwendbarkeit.

Mit anderen Worten: Hohe Kohäsion = klare Trennung, dh. eine Klasse soll nur eine bestimmte Aufgabe erfüllen.

Kopplung

Sie beschreibt, wie stark Klassen miteinander in Verbindung stehen. Ziel sollte es folglich sein, eine möglichst lose Kopplung, also wenig Abhängigkeiten untereinander zu erreichen. (Durch Datenkapselung und Zugriffsmethoden)

Statische Methoden und Attribute

Eigenschaften und Methoden, welche unabhängig von einem Objekt-Zustand (genauer, Objekt-Instanz) sind, sollten eigentlich über die Klasse, nicht über ein Objekt aufgerufen werden können -> Statische Dinge.

main

Die Main-Methode muss ebenfalls static sein. Wobei hier angemerkt sein sollte, das jede Klasse, die eine

```
public static void main(String [] args);
```

Methode implementiert, auch von der Kommandozeile aufgerufen warden kann.

Statischer Initialiser

Er hat keinen Namen und keine Parameter, kann aber dazu verwendet werden, eine Klasse beim Programmausführung zu initialisieren:

```
static {
   instanceCounter = 0;
}
```

Notiz

An dieser Stelle sollten wir zwei Wörter definieren/klarstellen:

Objektmethoden/-variablen sind einem Objekt zugeordnet (nicht static)
Klassenmethoden/-variablen sind einer Klasse zugeordnet (static)

Objektmethoden können auf Objekt- und Klassenvariablen/-methoden zugreifen, Klassenmethoden hingegen nur auf Klassenvariablen/-methoden.

Wrapper-Klassen

Für jeden primitiven Datentyp gibt es in Java einen sogenannten Wrapper. Er kapselt die primitive Variable in eine objektorientierte Hülle.

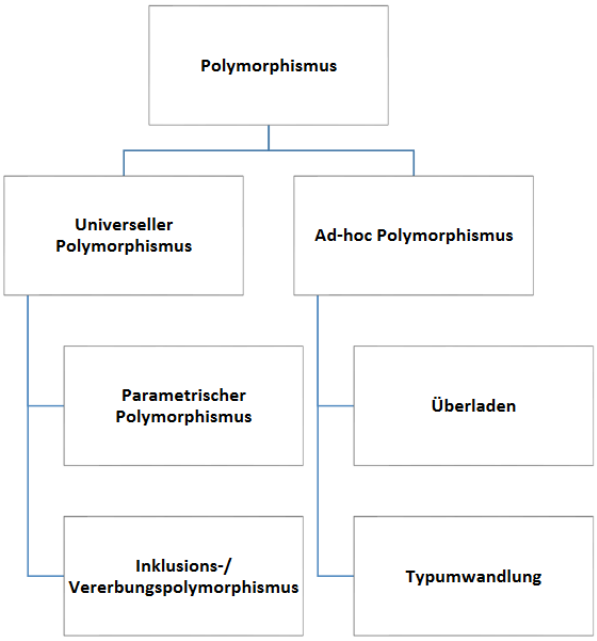
Der Grund dafür, Wrapper-Klassen bieten statische Hilfsmethoden an, z.b. zum Parsen oder Formatieren. Zusätzlich erhöht man die Datenkompatibiltät und Flexibität und Generics nehmen nur Objekte an.

Um auf die Wrapper-Klasse zuzugreifen, braucht man nur den primitiven Datentypen mit großem Buchstaben am Anfang schreiben :^}

Alle Wrapper-Klassen sind unveränderlich (final), weshalb sie nicht als vollständiger Datentyp gedacht sind. Jede Änderung führt zu neuem Objekt.

Autoboxing/-unboxing

Dabei führt der Compiler automatisch die Konvertierung zwischen primitiven Typ und Wrapper-Klasse durch. Damit soll erreicht werden, primitive Typen wie Referenztypen zu verwenden.




```
public static void mitAutoboxing(final int arg) {
    final Integer i = arg;    // Boxing
    final int j = i + 1;      // Unboxing
    System.out.println(i + " " + j);
}
```

Probleme

Führt allerdings zu vielen Problemen, wie z.b:

- Probleme bei == (kein Auto-Unboxing)
- Bei Boxing im Bereich von -128 bis +127 erhält man immer die gleiche Referenz
- Keine implizite Konvertierung
- Keine Konvertierung von null zu 0
- ...

Vererbung & Polymorphismus

Programme sollten in gewissen Situationen mit den verschiedenen Varianten (hiararchische Klassifikation) arbeiten, in bestimmten Situationen aber nicht unbedingten unterscheiden. Deshalb ist neben der Vererbung (welche die hierarchische Klassifikation vorgibt) auch der Polymorphismus unerlässlich.

Polymorphismus

Eine Variable oder eine Methode kann gleichzeitig mehrere Typen haben.

Dabei sind OOP-Sprachen immer polymorph, konventionelle dagegen monomorph.

Universeller Polymorphismus

Ein Name/Wert kann theoretisch **unendlich Typen** besitzen.

Ad-hoc-Polymorphismus

Ein Name/Wert kann nur **endlich viele verschiedene Typen** besitzen, wobei die Typen zur **Übersetzungszeit bekannt** sind.

Während wir bis zu diesem Abschnitt die Ad-hoc-Polymorphie (Typumwandlung, Überladen von Methoden) kennenlernenten, wurde der universelle Polymorphismus noch nicht besprochen. Dazu gehören Parametrischer Polymorphismus (Generische Programmierung) und vorallem **Inklusionspolymorphismus** (Ersetzsbarkeitsprinzip, Dynamische Polymorphie, Vererbung v. Spezifikationen/Implementierungen).

Klassen

Unterklasse/Oberklasse

Eine Klasse S ist eine Unterklasse der Klasse A,
wenn S die Spezifikationen von A erfüllt,
umgekehrt aber A nicht die Spezifikation von S.
Dann ist A eine Oberklasse von S

Ersetzbarkeit

Wenn B eine Unterklasse von A ist, dann können alle Instanzen von A durch ein Exemplar der Klasse B ersetzt werden, wobei weiterhin alle zugesicherten Eigenschaften der Klasse A gelten.

Kategoriesierung

Es gibt drei Arten von Klassen:

- Konkrete Klassen
  Daraus können Instanzen erzeugt werden. Für alle spezifizierten Operationen muss es auch eine Implementierung geben.
- Schnittstellen-Klassen
  Dies sind Interfaces, also reine Spezifikationen einer Menge von Operationen. Daraus können keine Instanzen erzeugt werden. Damit erreicht man eine klare Struktur für Konkrete Klassen, welche Interfaces implementieren und kann von diesen Klassen gewissen Methoden erwarten.
- Abstrakte Klassen
  Stellen für mindestens eine der spezifizierten Operationen (meist) keine Implementierung bereit. Auch daraus kann keine direkte Instanz erzeugt werden, eine Instanz einer Unterklasse kann aber sehr wohl die abstrakte Klasse als Typ annehmen. Sie ist also eine Zwischenstufe zw. Interfaces und konkrete Klassen.

Interfaces

Der Inhalt von Interfaces besteht aus Methoden-Spezifikationen (ohne Rumpf). Also wirklich nur die Signatur einer Methode. Diese Interfaces können dann von einer Klasse implementiert werden (mit dem Schlüsselwort „implements“).

```
Public class B implements A {}
```

- Methoden sind **alle abstrakt**
- Methoden sind **immer public**
- Keine Konstruktoren
- Variablen sind erlaubt, dann aber nur „public static final“ (also nur Konstanten)

Eine Klasse darf dabei mehrere Interfaces implementieren, auch darf ein Interface von mehreren Klassen implementiert werden. Dann muss aber die Klasse alle Spezifikationen des Interfaces erfüllen.

Genauso darf ein Interface auch von einem anderen Interface abgeleitet werden. Dazu verwenden wir das Schlüsselwort „extends“. Funktioniert gleich wie bei der Klassenvererbung.

Inklusionspolymorphismus

Angenommen zwei Klassen implementieren ein Interface „complex“, dann können alle Objekte dieser beiden Klassen eine beliebigen Variable vom Typ „complex“ zugewiesen werden.
Folglich kann eine Variable auf unterschiedliche Objekte zeigen. Dies nennt man **Inklusionspolymorphismus**.

Der Typ einer Variable dient also nur als Schnittstelle. Sobald die Klasse eines Objekts diese Schnittstelle erfüllt, kann sie der Variable zugewiesen werden. Beim Aufruf einer Operation wird dann die entsprechende Methode ausgewählt.

Bindung von Methoden

Frühe Bindung, also bereits zur Compile Zeit, nennt man Statische Bindung.

Späte Bindung, also erst zur Laufzeit nennt man, eh klar, dynamische Bindung.

Einsatz

Der Einsatz von Interfaces soll eine isolierte Entwicklung von Implementierungen ermöglichen. Eine Klasse, die ein Interface implementiert, kann direkt in einer Anwendung eingebunden werden, wenn diese nur Variablen vom Interface-Typen verwendet.

Abstrakte Klassen

Sind ein Mittelding zwischen Interfaces und Konkrete Klassen. Sie können nicht instanziert werden, sehr wohl aber Implementierungen zur Verfügung stellen. Auch können sie Objektvariablen und weitere Spezifikationen enthalten. Sie werden mit „abstract“ signiert.

Eine abstrakte Klasse darf ein Interface nicht vollständig implementieren (bzw. Eine Klasse die ein Interface implementiert, es aber nicht erfüllt, muss abstrakt sein).

Entgegen dem Interface darf eine abstrakte Klasse auch den Zugriffschutz protected und default verwenden und sogar Konstruktoren zur Verfügung stellen.

Subtyping = Wenn Klasse von Interface erbt
Subclassing = Wenn Klasse von anderen Klasse (Implemen.) erbt.

Vererbung

Überschreiben

Wird in einer Unterklasse eine Methode, welche mit dieser Signatur bereits in der Super-Klasse existiert, überschrieben, so entscheidet die Polymorphie, welche Methoden tatsächlich aufgerufen wird. Weil Java mit dynamischen Typen arbeitet, ist der tatsächliche Typ des Objekts entscheidend.

Eine mit final gezeichnete Methode kann nicht von den Unterklassen überschrieben werden.

Weiters gilt:

- Der Zugriffsschutz darf gelockert werden (z.B. protected -> public)
- Ergebnistyp darf abgeleitet werden
- Rumpf darf komplett ersetzt werden

Konstruktoren

Jeder Konstruktor einer Subklasse muss **immer zuerst** einen Superklassen-Konstruktor aufrufen. Dazu verwenden wir „super“

super() darf übrigens auch in normalen Methoden verwendet werden – z.B. zur reinen Erweiterung:

```
super.getSomething()
```

Downcast

Wird ein Objekt vom Basis-Typ A durch expliziten Cast einer Variable vom Typ B (der eine Subklasse von A ist) zugeweisen, nennt sich dies Downcast. Er ist problematisch, aber erlaubt.

Instanceof

Objektvariablen sind polymorph. Sie haben einen statischen Typ (Deklaration) und eine aktuelle Klassenzugehörigkeit (dynamischer Typ). Letzterer kann mittels instanceof abgefragt werden, wobei die Vererbung darin enthalten ist. Das heißt, eine Instanz der Klasse B (erbt von A) ist auch eine Instanz von A.

Kovarianz, Kontravarianz, Invarianz

Diese Begriffe kreisen um die Frage, was beim Überschreiben verändert werden darf.

Kovarianz (=entlang der Vererbungsrichtung)
Kontravarianz (=entlang der Vererbungsrichtung)
Invarianz (=Typen bleiben gleich)

Wobei damit immer die Redefinition v. Parameter- & Rückgabetypen gemeint sind.

In Java gilt bei Rückgabetypen eine Kovarianz, ansonsten immer Invarianz.
Aber was heißt das jetzt?

Kovarianter Rückgabetyp erlaubt einfach jeden kompatiblen Ergebnistyp bei der Redefinition geerbter Methoden und bei der Implementierung von Interfacemethoden. Wir dürfen also bei überschriebenen Methoden den Rückgabetypen nur „verschärfen“:

```
public class A{
  public A get(){...}
}

public class B extends A{
  public B get(){...}
}

public class C extends B{
  public C get(){...}
}
```

Delegation

Ein Objekt setzt eine Operation so um, dass der Aufruf der Operation an ein anderes Objekt delegiert (weitergerreicht) wird. Die Verantwortung einer Implementierung wird daher an eine andere Klasse weitergerreicht, wodurch Quellcode eingespart werden kann, wenn so Spezifikationen gemeinsam genutzt werden.

Zudem kann ein Delegat zur Laufzeit geändert werden, wodurch wir mehr Dynamik erhalten.

Problematiken und weitere Infos

Fragile Base Class Problem

Anpassungen an der Basisklasse können zu unerwarteten Verhalten von abgeleiteten Klassen führen → Schwierige Wartung

Gilt das Prinzip der Ersetzbarkeit auch in Zukunft?

Sind spätere Änderungen sehr wahrscheinlich? Dann lohnt sich eine Vererbung von Spezifikationen, möglicherweiße auch Delegation.

Typobjekte

Jedem Typ ist ein eindeutiges Typobjekt zugeordnet. Es kann mit obj.getClass() ermittelt werden.

Hashcode

Ist eine eindeutige Kennnummer für jedes Objekt. Beachte, dies ist nicht immer möglich!

Die Anforderung dabei ist: der Hashcode muss immer gleich bleiben, solange sich das Objekt nicht ändert. Wenn zudem zwei Objekte mittels equals als gleich erkannt werden, müssen sie den gleichen Hashcode produzieren, umgekehrt gilt das aber nicht.

Clone

Eine Klasse muss

- Interface Cloneable implementieren
- Eigene öffentliche Methode clone() implementieren
- In clone() eine Kopie des Superklassenobjekts mit super.clone() erzeugen
- Datenelemente veränderliche Klassen einzeln mit clone()-Aufrufen kopieren

Clone macht daher nur Sinn, wenn alle verwendeten Klassen clone() korrekt implementieren.

Java

Sprach-Informationen

Start war 1991 durch Sun Microsystems, welches seit 2010 ein Tochterunternehmen von Oracle ist. Java wurde angelehnt an C++, mit dem Ziel, Plattform-unabhängige Programme zu entwickeln und später auch für Internet-Programmierung angewandt werden zu können.

Durch Javas gemischtes Ausführungsschema wird der Quellcode zunächst über den Java Compiler in Java-Bytecode kompiliert. Dieser wird mithilfe von Bibliotheken auf der Java-Virtual Machine ausgeführt, welches die Befehle an das darunterliegende Betriebssystem weitergibt.

Der Bytecode ist also für alle Rechner/Betriebssysteme gleich, dafür aber gibt es für jede Plattform eine eigene JVM.

Sprach-Eigenarten

Bezeichner dürfen nicht mit Zahlen beginnen und keine Sonderzeichen enthalten.

In Java gibt es keinen Überlauf.

Stärke	Präzedenzgruppen	Assoziativität
14	( ), [ ], ., ++ (postfix), -- (postfix)	links
13	++ (präfix), -- (präfix), +(unär), -(unär), ~, !, (type), new	rechts
12	*, /, %	links
11	+, -, +(String-Konkatenation)	links
10	<<, >>, >>>	links
9	<, <=, >, >=, instanceof	links
8	==, !=, == (Referenz), != (Referenz)	links
7	& (bitweises UND), & (logisches UND)	links
6	^ (bitweises XOR), ^ (logisches XOR)	links
5	| (bitweises ODER), | (logisches ODER)	links
4	&& (logisches UND mit Short-Circuit-Evaluation)	links
3	|| (logisches ODER mit Short-Circuit-Evaluation)	links
2	?: (bedingte Auswertung)	rechts
1	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<==, >>==, >>>==	rechts

Dangling else

```
if (counter < 5)
  if (counter % 2 == 0)
    System.out.println("HERE1");
else
  System.out.println("HERE2");
```

Wobei counter == 7 zu keiner Ausgabe führt und counter == 3 zu „HERE2“

Switch

Switch-case ist bei Java mit short, byte, int, char oder seit java 7 mit String möglich.

Switch benötigt mehr Speicherplatz, arbeitet aber meist schneller als ifs.

Foreach

```
for(<Typ> variable: Collection)
{Anweisungsfolge; }
```

Final

Als final dürfen

Klassenvariablen
Objektvariablen
lokale Variablen
Methodenparameter
Methoden und Klassen

gekennzeichnet werden.

Damit gekennzeichnete Variablen oder Parameter dürfen maximal einmal einen Wert zugewiesen werden, danach sind sie Konstanten. **Wichtig**: Bei Variablen von Referenztypen bleibt nur die Referenz unverändert, nicht aber das Objekt.

Mit final gezeichnete Methoden dürfen nicht mehr von erbenden Klassen überschrieben werden. Auch Klassen dürfen mit final gekennzeichnet werden, dann sind alle darin enthaltenen Methodne final und es darf keine Subklasse erstellt werden.

Native Methoden

Methoden, die nicht von der JVM direkt unterstützt werden, welche Plattform-abhängig sind, können aus Java heraus aufgerufen werden und nennen sich native Methoden.

Annotations

Können als Metadaten angeführt werden und sind seit Java 1.5 dabei. Sie werden vom Compiler verarbeitet und enthalten Anweisungen für diesen (z.B.: @Test um eine Test-Methode zu markieren, oder @override für überschriebene Methoden)

Garbage-Collector

In Java brauchen wir uns nicht um unbenutzten Speicher, MemoryLeaks etc. kümmern. Dafür haben wir den Garbage-Collector, der automatisch unbenutzten Speichern sammelt und freigibt.

Vergleiche

Um zwei beliebige Elementtypen zu vergleichen, müssen wir als Programmierer in gewissen Fällen definieren, welches Element das „größere“ sei. Dazu können wir entweder das Interface „Comparable<T>“ implementieren, oder den java.util.Comparator verwenden.

Entscheiden wir uns für das generische Interface Comparable, so müssen wir eine Vergleichsmethode „compareTo“ implementieren, dessen Ergebnis

- > 0, wenn Objekt größer als sein Parameter
- < 0, wenn Objekt kleiner als sein Parameter
- 0 wenn die Objekte ident sind.

Der Comparator hingegen ermöglicht das Vergleichen nach verschiedenen Kriterien. Dafür sind viele Collection-Methoden mit einem zusätzlichen Parameter, dem Comparator, überladen.
Zum Vergleich wird einfach das übergebene Comparator-Objekt genutzt.

Der Comparator ist im Grunde ein generisches Interface „Comparator<T>“ mit einer Methode „compare“. Diese nimmt zwei Objekte und liefert obige ints zurück.

Datentypen

Alle Daten haben einen Datentyp, welcher die Benutzung der Daten einschränkt und den Wertebereich festlegt, bzw. Welche Operationen darauf angewandt werden dürfen.

Java ist **streng typisiert**, das bedeutet Datentyp und entsprechende Operationen sind während Programmlaufzeit unveränderlich. Sie müssen deklariert werden, damit der Compiler Speicher reservieren kann und auf Korrektheit überprüfen kann.

Primitive Datentypen

Sind nicht veränderbar oder ergänzbar, keine Methoden verfügbar und in Java beginnend mit Kleinbuchstaben.

Ganze Zahlen

Byte (8-Bit), short (16-Bit), int (32-Bit), long (64-Bit)

Referenztypen

Sind Klassen, in Java beginnend mit Großbuchstaben

Strings in Java

Strings sind Objekte, für die es sogar einen Literal („“) gibt. Auch + (und +=) sind erlaubt.

Strings sind jedoch nach dem Anlegen **immutable**, das heißt, sie können nicht mehr verändert werden. Eine Änderung erzeugt also implizit ein neues Objekt.

Verschiedene Konstruktoren sind möglich:

```
String s1 = "Hallo";
String s2 = new String();
String s3 = new String(s1);
char charArr[] = {'S','W','E','2'};
String s4 = new String(charArr);
String s5 = new String(charArr,1,2);
```

Würde hier noch ein neuer String s6 = „Hallo“ erzeugt werden, wird dafür kein neues Objekt angelegt, sondern das s1 vom Stringpool referenziert.

Gerade weil Strings also nicht sonderlich Performant sind, sollte man die Verwendung von StringBuilder/StringBuffer in Erwägung ziehen.

Reservierte Schlüsselwörter

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto (*)	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const (*)	float	native	super	while

Konstanten

Konstanten werden mit dem Schlüsselwort „final“ markiert. Sie müssen aber nicht sofort initialisiert werden. Es können damit nur Variablen und globale Variablen markiert werden:

```
final int integer = 10;
static final int value = 10;
```

Collections

Arrays

Sind Zusammenfassung gleichartiger Elemente, mit einer fixen Größe. Das heißt, beim Anlegen des Arrays wird die Größe festgelegt. Nur dynamische Arrays können nachträglich vergrößert werden (dafür gibts spezielle Klassen). Jede Arrayvariable ist in Java eine Referenzvariable (wie in C)!

Deklariert werden sie mittels dem „[]“ Postfix-Operator, z.b. int[] oder double[][]. Hierbei wurde allerdings noch kein Speicher alloziert, nur eine Referenzvariable angelegt. Um den Speicher zu reservieren (und die Größe damit festzulegen) verwenden wir einfach

```
int[] arr = new int[10];
```

Die Arrayelemente sind dann bereits initialisiert (mit Standardwert)! Um eine eigene Initialisierung vorzunehmen können wir folgendes machen:

```
int[] arr = {3, 4, 5, 6, 7};
arr = new int[]{3, 4, 5, 6, 7};
```

Die Arrayelemente werden über einen Index angesprochen, der vom Typ int sein muss (daher auch short, byte oder char), wobei der Indexbereich von 0 bis (arr.length - 1) geht.

```
arr[3] == 6
```

Referenz und Zuweisung

An dieser Stelle muss ich nochmal betonen: Arrayvariablen sind in Java Referenzvariablen! Das heißt, beim Zuweisen wird keine Kopie erstellt, sondern nur die Referenz kopiert:

```
int[] x = new int[10], y;
y = x;
y[1] = 7;
System.out.println(y[0] + " " + x[0]);
System.out.println(y[1] + " " + x[1]);
```

```
Ausgabe:
0 0
7 7
```

Um den reservierten Speicher eines Arrays freizugeben, reicht es die Referenz zu löschen – der Garbage Collector wird den Array danach nach Lust und Laune löschen.

Mehrdimensionale Arrays

Arrays können beliebig viele Dimensionen haben, wobei dann halt jedes Arrayelement selbst wieder ein Array ist. Genau so verhält es sich auch in der Sprache:

```
int[][] matrix = new int[3][3];
int[][] matrix2 = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

Zugriff logischweiße über:

```
x = matrix2[1][2];
```

Length auf matrix würde „3“ zurückgeben, matrix[0].length auch, aber nur weil drunter 3 Elemente liegen. Also gibt array.length immer nur die eigene (nicht „rekursiv“) an.

Mehrdimensionale Arrays müssen aber nicht zwangsläufig rechteckig sein. Die Letzte Dimension darf nämlich „offen“ bleiben und erst nachträglich initialisiert werden:

```
int[][] arr1 = new int[2][];
for (int i = 0; i < arr1.length; i++)
  arr1[i] = new int[i + 2];
```

Foreach

```
int[] arr = { 3, 4, 5, 6, 7 };
for (int b : arr)
  System.out.println(b);
```

JCF – Java Collection Framework

Die Motivation dahinter ist, dass wir sehr oft bestimmte Datenstrukturen benötigen. Diese sollten dann im Idealfall vorhanden und generisch sein, sowie einige Algorithmen dafür schon existieren. Genau dafür wurde das JCF gemacht.

Das Framework finden wir im Paket java.util
Dabei gibt es für unterschiedliche Datenstrukturen unterschiedliche Interfaces, die den Zugriff auf die Datenstruktur angeben. Aber erst eine konkrete Implementierung dieser Interfaces (welche natürlich frei gewählt und unterschiedlich sein können) ermöglicht das Verwenden.

Mit anderen Worten: Das JCF besteht in der Grundlage aus Interfaces, welche Basisoperationen, Mengenoperationen und Feldoperationen zur Verfügung stellen. Dadurch können wir konkrete Implementierungen jederzeit austauschen, ohne dabei das Programm in seinem Aufbau zu verändern.

Collection	• Eine Collection verwaltet Objekte (Elemente). • Interface enthält generelle Methoden (für alle Collection-Typen). • Es gibt **keine** direkte Implementierung dieses Interfaces.
Set	• Eine Collection die keine duplizierten Elemente enthält.
List	• Eine geordnete Collection (mit duplizierte Elementen) • Elemente können über einen Index angesprochen werden (nicht immer effizient).
Queue	• Verwaltung von Warteschlangen (ähnlich zu Listen) • FIFO, Prioritätswarteschlangen
Map	• Maps verwalten Schlüssel mit dazugehörigen Werten. • Schlüssel sind eindeutig. • Jeder Schlüssel kann einen dazugehörigen Wert haben.
SortedSet und SortedMap	• Sind spezielle Versionen von Set und Map, bei denen die Elemente (Schlüssel) in aufsteigender Reihenfolge verwaltet werden.

Dabei gibt es generelle Implementierungen für die Interfaces Set, List, Map und noch ein paar weitere.

Eigenschaften einiger ausgewählter Klassen

ArrayList
- Indexzugriff auf Elemente ist überall ungefähr gleich schnell.
- Einfügen und Löschen ist am Listenende schnell und wird mit wachsender Entfernung vom Listenende langsamer.

LinkedList
- Indexzugriff auf Elemente ist an den Enden schnell und wird mit der Entfernung von den Enden langsamer.
- Einfügen und Löschen ohne Indexzugriff ist überall gleich schnell.
- Ansonsten abhängig vom Indexzugriff.

HashSet
- null-Elemente sind zulässig.
- Einfügen, Suchen und Löschen sind immer gleich schnell.
- Aber: Rehashing kann zu Performance-Problemen führen (siehe auch API-Beschreibung).

TreeSet
- null-Elemente sind nicht erlaubt.
- Die Geschwindigkeit von Einfügen, Suchen und Löschen ist proportional zum Logarithmus der Anzahl der Elemente.
- Auf die Elemente eines TreeSets muss eine Ordnung definiert sein (müssen vergleichbar sein, d.h. das Interface `Comparable<T>` implementieren).

Übersicht wichtiger Klassen

Typ	Ordnung bleibt	Null-Elemente	Duplikate
ArrayList	ja	ja	ja
LinkedList	ja	ja	ja
HashSet	nein	ja	nein
TreeSet	ja	nein	nein
HashMap	nein	ja	Schlüssel nein, Werte ja
TreeMap	ja	Schlüssel nein, Werte ja	Schlüssel nein, Werte ja

Methoden

Für Listen und Sets gibt es einige gemeinsame Methoden, wie z.B.:

```
int size()
boolean add(T t)
boolean remove(T t)
boolean contains(T t)
T get(int i)
T set(int i, T t)
int indexOf(T t)
```

Maps

Maps sind im Allgemeinen Arrays, mit einem beliebigen Indextypen. Das heißt, statt die Arrayelemente mit Zahlen-Indizes anzusprechen, sprechen wir die Map-Elemente mit, z.B., einem String an. Folglich sind Maps KeyValue-Pairs.

Dabei ist jedem Schlüssel genau ein Wert zugeordnet, jedem Wert kann aber mehrere Schlüssel zugeordnet werden.

Wir unterscheiden zwischen HashMaps und TreeMaps, wobei HashMaps ungeordnet sind und TreeMaps geordnet und balanciert. Die HashMaps vergleichen den HashCode mittels equals, weshalb diese hashCode und equals implementiert haben sollten!

Obwohl Maps nicht das Collection-Interface implementieren, gehören sie zum JCF. Wir können beide daher verknüpfen.

Einige wichtige Methoden

Put, get, containsKey, containsValue, remove, aber vorallem:

keySet -> liefert Menge der Schlüssel als Set
values -> liefert Werte der Map als Collection, Ordnung unbekannt
entrySet -> liefert Menge der Einträge der Map
getKey, getValue

die obigen Methoden erzeugen aber keine neuen Collections, sondern sogenannte „views“, also Sichtweisen. Also sind die Methodne sehr effizient, weil der View einfach auf die zugrunde liegende Map zugreift. Änderungen an einer Sicht wirkt sich auf die Map aus, Änderungen an der Map auf alle Sichten.

Maps kennen keine Iteratoren, Sets dagegen schon. Wir können wir sehr wohl über Maps iterieren, müssen aber einen „Umweg“ gehen.

Iteratoren

Iteratoren bieten meist eine Methode an, mit der sie das nächste Element in der zu durchlaufenden Collection anbieten. Sie iterieren also über eine Collection. Grundlegend gibt es dabei 3 Methoden:

- hasNext
- next
- remove

Alle Iteratoren müssen das generische Interface „Iterator<T>“ implementieren und den gleichen Elementtyp wie die zugrunde liegende Collection besitzen.

Der Iterator läuft von Beginn an, Element für Element, bis keine Elemente mehr übrig sind, dann ist er verbraucht. Wollen wir nochmal durchfahren, müssen wir einen neuen Iterator erzeugen.

Eigenschaften/Modifikationen

Wird eine Collection modifiziert (neues Element zb), so werden alle Iteratoren dieser Collection ungültig, ein Zugriffsversuch über den Iterator endet in einer ConcurrentModificationException. Man nennt dies auch fail-fast, weil der Iterator sofort unbrauchbar gemacht wurde.

Wichtig ist aber zu wissen, dass nur strukturelle Änderungen der Collection ein fail-fast bewirken – das reine Ersetzen eines Elements zieht keine Strukturelle Änderung nach sich und lässt den Iterator damit intakt.

Um nun (strukturelle) Modifikationen trotz der Existenz von Iteratoren zu ermöglichen, bietet meist der Iterator selbst eine Änderungsoperation an, über welche die Collection modifiziert werden kann, ohne den Iterator zu zerstören. So definieren zB alle Iteratoren die Methode „remove“, welche das zuletzt überquerte Element aus der Collection entfernt.

Hierbei sei angemerkt, Änderungen über Iteratoren bewirken dennoch einen fail-fast bei allen anderen Iteratoren.

Listen-Iterator

Listen haben ein eigenes generisches Iterator-Interface „ListIterator<T>“. Diese müssen sich vorwärts und rückwärts bewegen können, folglich ist ein ListIterator auch nicht verbraucht, wenn er an eines der beiden Enden angekommen ist.

Listen verfügen neben „remove“ auch noch über „add“ und „set“

Methoden

DRY
Don’t Repeat Yourself!

Arten von Unterprogrammen

Unterscheidung nach Aufgabe

Prozeduren

Sind Unterprogramme, die Anweisungen ausführen, aber keinen Rückgabewert haben. Sie dürfen Nebeneffekte haben (zb. Klassenvariablen ändern)

Funktionen

Unterprogramme, die Anweisungen ausführen und beim Rücksprung einen Wert an das aufrufende Programm übergeben. Sie sollten keine Variable außerhalb der Methode verändern.

Unterscheidung nach Zugehörigkeit

Unterprogramme

Eigenständiger Teil des Programms, C-Funktion, Nicht in Java!

Methoden

Gehören einer Klasse an, zb. Java-Methoden

Signatur

Die Signatur einer Methode (in Java) besteht aus dem Namen der Methode + der Liste der Typen der formalen Parameter (Namen der Parameter nicht). Daraus folgt, dass in Java der Rückgabetyp nicht zur Signatur gehört, sehr wohl aber die Reihenfolge der Parameter.

Parameterübergabe

Java ist call-by-value, das heißt, bei primitiven Datentypen wird **immer** eine Kopie des aktuellen Parameters übergeben, bei Objekten eine Kopie der Referenzvariable.

Das heißt, bei primitive Datentypen kann der aktuelle Parameter in der aufrufenden Methode nicht verändert werden, bei Referenzobjekten kann die Referenz nicht verändert werden (sehr wohl aber das referenzierte Objekt)

Varargs-Parameter

```
Public static int sum(int... values){}
```

Der Varargs-Parameter (hence, "..." nach dem int) ermöglicht eine beliebige Anzahl von Paramtern eines Typs, welche nach dem Aufruf zu einem Array umgewandelt werden.

Globale Variablen

Werden durch static gekennzeichnet und werden bei Programmstart reserviert. Sie existieren über den Methodenaufruf hinweg.

Javadoc

Bei herkömmlichen Programmen sind Quelltext und Dokumentation in verschiedenen Dateien, weshalb natürlich eine Inkonsistenz auftreten kann. Die Javadoc soll dies verhindern.

```
/**
* ... hier ein javadoc-text
*/
```

Javadocs stehen immer vor der Definition von Klassen/Interfaces, Methoden, Datenelementen und kann aus einem einfachen Satz, einer ausführlichen Beschreibung und einer Liste von Tags bestehen.

Hierbei wird auf die offizielle Doku verwiesen:
http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Konventionen

Englische Sprache verwenden, aussagekräftige Kommentare und besser zu viel als zu wenig dokumentiert. Der Leser des Kommentars soll wissen, was die Methode tut, welche Parameter sie erwartet, was sie zurückgibt und welche Exceptions auftreten können.

Tags

Sie markieren Informationen mit einer bestimmten Bedeutung. Dabei bestehen sie aus einem „@" und einem Schlüsselwort, wobei nach dem Text noch eine Beschreibung folgt. Jeder Tag muss in einer neuen Zeile stehen.

Während diese Tags bei Klassen z.b

- @author Name
- @version 1.0

- @since jdk-version
- @see reference

Sein können, **muss** (naja, eig sollte) die Javadoc bei Methoden folgende enthalten

- @param paramName infoText
- @return infoText
- @throws exceptionClass text

Natürlich gibts da noch viel mehr Tags:
http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

HTML

Der Inhalt von Doc-Kommentaren wird in HTML-Dateien übersetzt, weshalb man auch HTML-Tags verwenden kann (wovon aber im allgemeinen Fall abgeraten wird). Wenn, dann sollten Sie sinnvoll sein:

```
@author Sebastian Holzer <a href=mailto:meine-email@uibk.ac.at> meine-email@uibk.ac.at</a>
```

Compiler

Die Javadoc hat einen eigenen Compiler (javadoc), der die Doku in das HTML-Format kompiliert. Diese Doku wird von javac (dem Java-Compiler) ignoriert.

Schalter:

-d path	Alle generierten HTML-Seiten werden im Directory path abgelegt.
-public	Nur public-Elemente werden in die Dokumentation aufgenommen.
-author	Übernimmt das @author-Tag (nicht default).
-version	Übernimmt das @version-Tag (nicht default).
-help	Gibt eine Liste der Schalter von Javadoc aus.

Exceptions

//TOODOOOOOOOO Folie 8

Wichtige Code-Snippets

Ausgabe

```
System.out.println();
```

Wobei out ein Stream ist.

String-Methoden

```
public class StringTest {
    public static void main(final String[] args) {
        final String s1 = "Das ist ein spezieller Test";
        System.out.println(s1.length());
        System.out.println(s1.charAt(0));
        System.out.println(s1.indexOf('i'));
        System.out.println(s1.lastIndexOf('i'));
        System.out.println(s1.indexOf('i', 5));
        System.out.println(s1.lastIndexOf('i', 10));
        System.out.println(s1.indexOf("ei"));
        System.out.println(s1.substring(8));
        System.out.println(s1.substring(4, 15));
    }
}
```

Ausgabe:

```
27
D
4
16
9
9
8
ein spezieller Test
ist ein spe
```

Equals

Eine korrekte Implementierung der Equals-Methode erfüllt folgende Regeln:

- Reflexivität: x.equals(x) -> true
- Symmetrie: x.equals(y) == y.equals(x)
- Transivität: x.equals(y) -> true, y. equals(z) -> true, dann gilt auch x.equals(z) -> true
- Konsistenz: Zwei Objekte müssen bei wiederhoteln Aufrufen immer das gleiche Ergebnis liefern.
- Für alle Objekte x, die nicht gleich null sind, gilt: x.equals(null) -> false