Zum Thema Objekte:

Was macht eigentlich Objekte aus? In „klassischen“ prozeduralen Programmiersprachen wie Pascal haben wir eine klare Trennung zwischen der Modellierung von Informationen und der Modellierung der Verarbeitung. Die Informationen sind in Form globaler Zustände von Variablen bzw. deren Werten modelliert, die Verarbeitung ist die (schrittweise) Veränderung des globalen Zustands; eine Programmausführung ist eine Überführung von Daten von einem Anfangs- in einen Endzustand.

OOP betrachtet eine Programmausführung als System kooperierender Objekte. Diese haben jeweils ihren eigenen lokalen Zustand und verfügen über die Fähigkeit, auf Nachrichten zu reagieren, indem sie ihren Zustand ändern und/oder Auskunft über diesen geben. Ein Objekt hat eine eigene Identität und eine Lebensdauer. → Analogie zu Gegenständen der materiellen Welt. Ein Objekt hat bestimmte Eigenschaften (Attribute) und eine Schnittstelle, bestehend aus den Nachrichten, die es versteht.

Objekte

- haben einen Zustand
- haben eine Identität
- können Nachrichten verarbeiten, indem sie Methoden ausführen
- sind theoretisch (die Praxis real existierender OO-Sprachen sieht anders aus; wir werden dies noch erörtern) selbstständige aktive Einheiten, die unabhängig und parallel agieren (Inhärente Parallelität des OO-Paradigmas)
- bilden eine Einheit aus Daten und den auf ihnen definierten Operationen
- modellieren (abstrahierte) Ausschnitte der realen Welt
- haben eine Schnittstelle, welche die Möglichkeiten beschreibt, mit dem Objekt zu interagieren
- können über Referenzen angesprochen werden, dabei kann es mehrere Referenzen auf dasselbe Objekt geben (Aliase). Referenzen können in Variablen abgelegt werden
- können mit anderen Objekte verbunden sein, so dass die Existenz der Verbindung Teil des
- Objektzustands ist (→ Attribut von Objekt A enthält Referenz auf Objekt B)

Als Reaktion auf eine Nachricht führt ein Objekt eine seiner Methoden aus. Was dabei genau geschieht, liegt in der Verantwortung des Objekts, ist Teil seines „Wesens“. Als Folge einer Nachricht kann ein Objekt:

- seinen Zustand ändern
- Auskunft über seinen Zustand geben
- Nachrichten an andere Objekte schicken
- Objekte erzeugen oder löschen

Prozedurale Denkweise: Man sucht primär eine *Methode*, die etwas bestimmtes kann und versucht, diese irgendwie aufzurufen. Realwelt: In dieser würde man niemals versuchen, eine "globale Methode" aufzurufen: Wenn etwas getan werden muss, soll es normalerweise *an einem bestimmten bekannten Objekt* getan werden, dessen Zustand sich dadurch ändern soll (Beispiel: Motorrad starten). Manchmal benötigt man dazu weitere Objekte, welche die Fähigkeit besitzen, das zu ändernde Objekt zu manipulieren (Beispiel: Handwerker). Zur Kommunikation mit einem Realwelt-Objekt benötigt man ebenfalls eine Referenz (Telefonnummer, Adresse, Kontonummer...). → "Die Realwelt ist objektorientiert."

Literale

... sind textuelle Repräsentationen von Objekten, der einfachste Weg, ein Objekt zu beschreiben und letztlich seine Erzeugung zu veranlassen. Typische Beispiele: Zahlen, Zeichen, je nach Sprache String-Literale, Array-Literale, Symbolliterale, sogar Blöcke... Beispiele (Smalltalk):

Zeichen: `$a`
Zahlen: `42 -7`
String: `'Test'`
Symbol: `#Smalltalk`
Array: `#(1 2 3)`
Block: `[:x | x < 0 ifTrue: [Transcript show: 'Fehler']]`

Gleichheit vs. Identität

Smalltalk: Test auf Gleichheit mit =, Test auf Identität mit ==

Variablen

Wertsemantik vs. Verweissemantik, Zusammenhang mit unveränderlichen Objekten, Zusammenhang mit der Frage Gleichheit vs. Identität.
Lebensdauer Variable != Lebensdauer Objekt (Aliase!)

Aliasing

Problem: Änderung über einen Alias ändert das Objekt → Quelle für Programmierfahler.
Problem: Sichtbarkeit ist typischerweise der Ebene der Variablen geregelt, nicht auf der der Objekte. Aliase hebeln diesen Schutz aus. Beispiel: Liste mit verketteten Entry-Objekten: Hat jmd. einen Alias auf ein Entry-Objekt, kann er die Verkettung ändern und damit die Invarianten der Liste ungültig machen.

Objektzustand: Instanzvariablen

Instanzvariablen (Attribute) repräsentieren den Zustand eines Objekts in Form von Beziehungen zu anderen Objekten (je nach Sprache: auch primitive "Werte").
Smalltalk: Benannte Instanzvariablen (1:1) vs. indizierte Instanzvariablen (1:n). Problem bei indizierten: Keine Unterscheidbarkeit. Was man will: Benannte 1:n-Beziehung. Lösung: Zwischenobjekte. Vorteil einer Repräsentation der 1:n-Beziehung über Zwischenobjekte: Die "Beziehung selbst" ist Objekt und versteht Nachrichten, kann also je nach Nachricht verschiedene "Sichten" auf die Beziehung liefern, z.B. sortierte Sicht. Außerdem können unterschiedliche Arten von Zwischenobjekten bestimmte Eigenschaften der durch sie repräsentierten Beziehung garantieren, z.B. die, dass auf der n-Seite der Beziehung von einander gleichen Objekte nur höchstens eine vorkommen darf (Menge).

Ausdrücke, Nachrichten und Kontrollstrukturen

Ausdrücke:

Ausdrücke stehen für Objekte, liefern bei "Auswertung" ein Objekt bzw. eine Objektreferenz.

- Literale (primitivste Form)
- Zuweisung: Bewirkt neuen Wert einer Variablen (das kann einen Zustandswechsel eines Objekts bedeuten, wenn die Variable ein Attribut ist), hat/liefert selbst einen "Wert", nämlich das zugewiesene Objekt.
- Nachrichtenausdrücke

Nachrichtenausdrücke

- drei Arten:
  - unäre Nachrichten: Nachrichten ohne Parameter (außer dem "impliziten Parameter", dem Empfängerobjekt). Beispiel: `lichtSchalter schalteLichtEin`
  - binäre Nachrichten: Nachrichten mit einem Parameter. Beispiel: `1 + 2`
    Hier wird dem Objekt 1 die Nachricht + geschickt, mit dem Objekt 2 als Parameter.
    Syntaktische Besonderheit: Bestehen aus einem speziellen Zeichen, ohne : zur Angabe des Parameters. Obiges Beispiel könnte auch so aussehen: `1 plus: 2`
  - Schlüsselwortnachrichten: Alle anderen nichtunären Nachrichten. Diese können aus mehreren Teilen bestehen. Beispiel: `12 printOn: Transcript`
- Nachrichtenselektor: *Ein* Symbol, welches die Nachricht identifiziert. Im letzten Beispiel ist der Nachrichtenselektor: `tueEtwasMit:und:`
- Nachrichtenausdruck besteht also aus Empfängerausdruck, Nachrichtenselektor und Argumentausdrücke. Möglichkeit der Schachtelung: Argumentausdrücke und Empfängerausdruck können ja selbst wieder Nachrichtenausdrücke sein.
- Ein Nachrichtenausdruck gibt immer etwas zurück (im Ggs. etwa zu Prozeduren in Pascal oder void-Methoden in Java), wenn es keinen sinnvollen Rückgabewert gibt, ist es Standard, den Nachrichtenempfänger selbst zurückzugeben.
  Auswertungsreihenfolge von Nachrichten: Grundsätzlich von links nach rechts, aber: Vorrang unärer vor binären vor Schlüsselwortnachrichten. Achtung: Bei mehrstelligen Schlüsselwortnachrichten werden alle Schlüsselwörter als zu einer Nachricht gehörig angesehen.
  Siehe Codebeispiel Pharo, Kommentar der Klasse Person.
  Zuweisung und Return-Anweisung sind keine Nachrichten, sie werden jeweils zuletzt ausgeführt, d.h. es wird erst der Ausdruck rechts vom Zuweisungs- bzw. Return-Operator ausgewertet.

Nachricht - Methode

Auswertung einer Nachricht an ein Objekt erfolgt durch dessen Methoden. Methodendefinitionen spezifizieren das Verhalten von Objekten. Methoden sind immer einem Objekt (Nachrichtenempfänger) zugeordnet, den Katalog der Methoden eines Objekts nennt man auch dessen Protokoll.

Zum Aufbau einer Methodendeklaration siehe Codebeispiel Pharo, Methoden der Klasse Person. Beachte dabei auch die implizite Rückgabe von self als Defaultverhalten bei allen Methoden außer asString, sowie den Zugriff auf Instanzvariablen durch die Methoden.

Parameterübergabe, Rückgabewert

Die aktuellen Parameter des Methodenaufrufs werden per impliziter Zuweisung an die formalen Parameter übergeben. Erläuterung CbR/CbV: In Smalltalk (wie in Java) nur CbV, wobei bedacht werden muss, dass beim "Übergeben von Objekten" Referenzen übergeben werden (siehe auch 1618-FAQ)! Am Ende der Methodenausführung wird der Rückgabewert implizit an den Nachrichtenausdruck (Methodenaufruf) übergeben. Diese Sichtweise ist berechtigt, schließlich steht dieser wie eine Variable als Platzhalter für das Ergebnis.

Dynamik

Unterschied Methode – Prozedur: Methodenaufruf hängt auch vom Empfängerobjekt ab. Das Ansprechen eines Objektes erfolgt über einen Ausdruck, dessen Wert eine Objektreferenz ist, z.B. eine entsprechende Variable. Ein solcher Ausdruck kann zu verschiedenen Zeitpunkten Referenzen auf verschiedene Objekte liefern, z.B. kann ein Objekt während seiner Lebenszeit durch Zuweisung von Referenzen an seine Attribute Verbindungen zu anderen Objekten eingehen und lösen. Daraus ergibt sich, dass erst bei der tatsächlichen Auswertung eines Nachrichtenausdrucks zur Laufzeit festgestellt werden kann, welches Objekt eigentlich der Nachrichtenempfänger ist! Selbst eine Analyse einzelner Codeabschnitte kann aufgrund von Parallelität problematisch sein.

Bindung = Zuordnung einer Anweisung im Code zu dem, was als Folge tatsächlich passiert:
Methodenaufruf → tatsächlich ausgeführte Methode
Attributselektion → tatsächlich gewähltes Attribut

Statische Bindung: Bei der Übersetzung ist klar, welcher Code als Folge eines Prozeduraufrufs ausgeführt wird. Eine entsprechende Sprungadresse kann ermittelt werden und wird – meist relativ zu einer Startadresse – direkt in den Maschinencode geschrieben. Normalfall bei klassischer prozeduraler Programmierung.

Dynamische Bindung: Aufgrund des o.g. Sachverhalts ist zur Übersetzungszeit nicht generell entscheidbar, welcher Code als Folge eines bestimmten Nachrichtenversands (Methodenaufrufs) ausgeführt wird! Die endgültige „Bindung“ erfolgt erst zur Laufzeit durch ein spezielles Ausführungsprogramm (Laufzeitumgebung, bei Java u.ä.: Virtual Machine, Runtime Engine usw). Normalfall bei objektorientierter Programmierung.

Zusammenhang Kontrollstrukturen – dynamische Methodenwahl

Programmablauf aus objektorientierter Sicht: Zustandswechsel von Objekten, "Verhalten". Steuerung des Programmablaufs durch Kontrollstrukturen. "Klassische" Kontrollstrukturen sind bekannt aus prozeduraler Programmierung (Sequenz, Verzweigung, Wiederholung, Unterprogrammaufruf). Wichtigste Kontrollstruktur der OOP: Dynamisch gebundener Methodenaufruf! Außer der Sequenz lassen

sich alle anderen "klassischen" Kontrollstrukturen durch diesen ersetzten... genau dies macht Smalltalk.

Codebeispiel Pharo Klassen True und False und deren Methoden ifTrue und ifFalse

Die Existenz eines klar definierten Protokolls ermöglicht Datenkapselung und Klassifizierung:

- Objekte haben die Kontrolle über „ihre“ Daten, sprich ihren Zustand, sie können die Konsistenz zwischen den Zuständen ihrer Attribute gewährleisten und Implementierungsdetails verbergen (Smalltalk: Instanzvariablen nach außen unsichtbar).
- Objekte können nach ihrer Schnittstelle klassifiziert werden: Objekte mit gleicher Schnittstelle bilden eine Klasse, wobei dieser Begriff hier nicht unbedingt 1:1 im Sinne z.B. einer Klasse einer Programmiersprache zu verstehen ist, auch wenn es natürlich einen Zusammenhang gibt.
- Naheliegende Umsetzung: Schnittstellenbeschreibung nicht pro Objekt, sondern pro Klasse, dazu später mehr.

Zusammenhang Nachrichtenmodell → Erweiterbarkeit:

Verschiedene Objekte können in der Lage sein, die gleiche Nachricht zu verstehen, und in jeweils spezifischer Weise zu reagieren. Ein Objekt, welches verschiedenen anderen Objekten eine Nachlicht m sendet, muss nicht angepasst werden, wenn zu diesen Objekten eine neue Sorte hinzukommt, so lange die Objekte dieser Sorte nur ebenfalls m verstehen.

Beispiel Pascal: Hier haben wir keine Zuordnung von Prozeduren zu unterschiedlichen Datenstrukturen. Eine Anpassung muss durch Fallunterscheidung, eine Erweiterung durch Hinzufügen weiterer Fälle erfolgen.

Hingegen OOP: Zuordnung erfolgt implizit durch den Nachrichtenmechanismus: Es wird eben genau der Code ausgeführt, den der Nachrichtenempfänger für die betreffende Nachricht vorsieht. Eigentlich unglaublich trivial und intuitiv verständlich, wenn man in Nachrichten denkt, die Objekten geschickt werden und die „prozedurale“ Trennung in Datenstrukturen und Verarbeitung vergisst.

Nicht vorhandene Methoden?

Smalltalk: Jedem Objekt kann jede Nachricht gesendet werden... was passiert, wenn das Objekt keine entsprechende Methode hat? Antwort: Die VM schickt dem ursprünglichen Empfänger der problematischen Nachricht eine spezielle Nachricht `doesNotUnderstand`: wobei sie den problematischen Nachrichtenselektor als Argument übergibt. Jedes Objekt kann seine Methode `doesNotUnderstand` so anpassen, dass (mehr oder weniger) sinnvoll auf den Fehler reagiert wird.
Vorgriff: Motivation für Typsysteme ist u.a., eben dieses Problem zu vermeiden, also sicherzustellen, dass einem Objekt nur solche Nachrichten gesendet werden, welche es auch versteht.

Blöcke, Iterieren über Collections

Noch ein bisschen Smalltalk... zunächst Blöcke:

Ein Block ist ein unbenanntes Objekt (Gegensatz zur benannten Methode), welches für eine Sequenz von Anweisungen steht (literal definiert, nämlich als Programmtext!). Ein Block kann ausge-

führt werden, indem man ihm die Nachricht `value` sendet. Blöcken können ebenso wie Methoden Parameter übergeben werden.

Ein Block hat einen sog. Home Contex, das ist die Methode, in welcher der Block deklariert wurde. Die Ausführung eines Blocks erfolgt im "lebenden Kontext" seiner Deklaration. Das bedeutet u.a., dass die freien Variablen des Blocks eben die sind, welche in seinem Home Context existieren (Smalltalk-Blöcke entsprechen Closures).
Codebeispiel Pharo Klasse Closures.Test, Methode test: (siehe Methodenkommentar)

Zum Home Context eines Blocks gehört auch dessen Aufruf-Stack. Eine explizite Return-Anweisung in einem Block sorgt deshalb dafür, dass nicht nur die Ausführung des Blocks beendet wird, sondern aus dem Home Context zurückgekehrt wird! (Continuation).
Codebeispiel Pharo Klasse BlockKontextTest. Dieses Beispiel ist ziemlich unsinnig, es zeigt lediglich die Vorgehensweise. Sinnvolles Beispiel siehe weiter unten

Diese Gestaltung von Blöcken erlaubt zusammen mit dynamischer Bindung die Implementierung von Kontrollstrukturen in Smalltalk selbst. Beispiele für if/else:
Codebeispiel Pharo Klassen True und False und deren Methoden ifTrue: und ifFalse:, sowie ifTrue:ifFalse: und ifFalse:ifTrue: denen jeweils Blöcke übergeben werden.

Codebeispiel Pharo Klasse Wundersam, Methode nextValue: (siehe Methodenkommentar)

Continuations erlauben z.B. eine Rückkehr aus "tiefen" Methodenaufruf-Ebenen. Mögliches Problem: Der Home Context könnte schon vor dem Zeitpunkt der Ausführung des Blocks zurückgekehrt sein. Dann liegt natürlich keine Rücksprungadresse mehr für ihn auf dem Stack → Fehler, siehe Beispiel S. 63 im Kurstext.

Beispiel für whileTrue:
Codebeispiel Pharo Klasse Wundersam, Methode teste, siehe auch Implementierung von whileTrue: in der Klasse BlockClosure, Protokoll "controlling"... Rekursion vom feinsten.

Beispiel für do:

```
#(5 3 1) do: [:i | Transcript show: i printString]
```

Codebeispiel Pharo, Klasse Collection. Wie man sieht, ist die Methode in Collection abstrakt. Die Methode do: für Array findet sich in SequenceableCollection, von der die Klasse Array sie erbt. Diese Methode verwendet intern die Methode to:do: aber welcher Klasse? Nun, begonnen wird mit 1, das ist ein Exemplar von SmallInteger, schauen wir uns also das Protokoll dieser Klasse an. Wie man sieht, wird to:do: von Number geerbt. Einfach mal reinschauen... wenn Sie der Unterstrich verwirt: Das ist lediglich eine alternative Form des Zuweisungsoperators :=

Über einen Array haben wir eben iteriert, das geht über andere Collections natürlich genauso. Interessanter sind aber die zusätzlichen Möglichkeiten der Iteration, welche diese "Zwischenobjekte" bieten.
Beispiele für select: reject: collect: detect finden sich im Kurs auf den Seiten 69 bis 71. Ein Beispiel analysieren wir hier.
Codebeispiel Pharo, Klassen FreundeBeispiel und Freund

```
freunde
    inject: true
```

```
into: [ :einsam :freund | einsam and: [freund eng not]]
```

Was geschieht hier? Wir haben eine Collection von Freunden, welche die Nachricht "eng" verstehen und mit true oder false beantworten. Der Block wird für jeden Freund in der Collection ausgeführt, wobei in der Blockvariablen `freund` immer das aktuelle Element der Collection landet.

Diesem wird die Nachricht `eng` geschickt, das Ergebnis wird verneint und mit dem aktuellen Wert von `einsam` (diese Blockvariable wird am Anfang mit dem ersten Parameter von `inject:into:` initialisiert) und-verknüpft. Das Ergebnis dieser Und-Verknüfpung ist dann der Wert von `einsam` für den nächsten Durchlauf der Iteration über die Elemente von `freunde`. In `einsam` steht also in jedem weiteren Durchlauf genau dann true, wenn das Ergebnis des vorigen Durchlaufs true war und der vorige Freund kein enger Freund war. Taucht einmal ein enger Freund auf, wird `einsam` im nächsten Durchlauf false und bleibt es logischerweise dann auch.

Klassen

- Klassifikation von Objekten durch Zuordnung zu einer Klasse (einem Allgemeinbegriff)
- Instanzbeziehung: Objekt x ist Instanz/Exemplar der Klasse Y, etwa "Peter ist eine Person". Damit ist klar, dass alles, was für Personen im Allgemeinen gilt, auch für Peter gilt.
- Es bietet sich an, Aussagen nicht pro Objekt zu treffen, sondern pro Klasse. In unserem Kontext natürlich besonders interessant: Aussagen über die Repräsentation der potentiellen Objektzustände (also die Instanzvariablen) und über das Protokoll der Objekte. Und so ist es denn auch der OO-Normalfall: Klassen bilden die Vorlagen für Objekte (= klassenbasierte Form der Objektorientierung, andere Möglichkeit wäre z.B. prototypbasiert).
- Extension einer Klasse: Menge der Objekte, die unter diese Klasse fallen
- Intension einer Klasse: Summe der Merkmale, welche eine Klasse ausmachen (Auswahlprädikat). Eine Klassendefinition liefert also die Intension der Klasse.
- Mit wachsender Intension einer Klasse schrumpft die Extension... klar: Je mehr Bedingungen an die Zugehörigkeit zu einer Klasse gestellt werden, desto weniger Objekte erfüllen diese.
- Instanziierung einer Klasse: Erzeugung eines Exemplars, funktioniert in Smalltalk über Methoden *der Klasse*, die selbst auch ein Objekt ist!

In Smalltalk:

- Metaklassen: Zu jeder Klasse gehört eine Metaklasse, von der die Klasse Exemplar ist. Metaklassen werden automatisch vom System erzeugt, wenn der Programmierer eine Klasse deklariert. Methoden der Metaklasse sind Klassenmethoden der entsprechenden Klasse.
- Konstruktoren sind in Smalltalk ganz normale Klassenmethoden.
- Klassenmethoden wie `new` haben keinen Zugriff auf Instanzvariablen des Exemplars, das sie erzeugen, benötigen also ggf. Zugriffsmethoden. Problem: Über diese Methoden wären die Instanzvariablen öffentlich zugänglich! Typische Vorgehensweise: *Eine* Instanzmethode `initialize` initialisiert das Objekt. Aufbau eines Standardkonstruktors wäre also:

  ```
  <Konstruktorname>
      ^ self new initialize
  ```

  Codebeispiel Pharo: "Konstruktoren" engerFreund und weiterFreund der Klasse Freund
- Da Konstruktoren ganz normale Klassenmethoden sind, müssen sie nicht zwangsläufig ein Exemplar ihrer Klasse zurückliefern. Interessant in Zusammenhang mit sog. Factorymethoden, welche je nach Bedarf Exemplare von Subklassen liefern!

Wie werden nun Klassen in Smalltalk letztlich erzeugt? Über den Klassenbrowser... aber wie macht der das? Antwort: Man schickt der vorgesehenen Superklasse der neuen Klasse eine entsprechende Nachricht.
Codebeispiel Pharo: Die besagte Nachricht entspricht dem, was man im Klassenbrowser für die Klasse selbst eingetragen sieht, siehe etwa die Klasse Student.

Klassenhierarchie, Generalisierung, Spezialisierung

Offenbar existiert also eine Hierarchie von Klassen, wobei die Standard-Superklasse für neue Klassen die Klasse Object ist.

- Eine Superklasse ist Generalisierung (Abstraktion) ihrer Subklassen, d.h. sie fasst die gemeinsamen Eigenschaften der Exemplare der Subklassen zusammen.
- Es bietet sich natürlich an, die Intension der Generalisierung nur einmal anzugeben, eben in der Superklasse. Subklassen müssen dann nur noch angeben, welche Klasse ihre Generalisierung ist.
- Gegensatz zur Generalisierung: Spezialisierung. Eine Subklasse ist Spezialisierung ihrer Superklasse.
- Student ist Spezialisierung von Person ist Spezialisierung von Object. Damit ist jeder Student auch eine Person und ein Objekt.

Vererbung, Hinzufügen, Überschreiben und Löschen von Methoden

- Vererbung ist ein Mechanismus, der dafür sorgt, die Eigenschaften und Methoden einer Generalisierung auf ihre Spezialisierungen zu übertragen.
- Eine Spezialisierung kann Eigenschaften und Methoden hinzufügen: Erweiterung der Intension.
- Eine Spezialisierung kann Methoden überschreiben, d.h. neu definieren.

Codebeispiel Pharo: Die Klasse Student fügt zu Person die Eigenschaft hinzu, eine Matrikelnummer zu haben, sie ergänzt eine Methode, welche diese Matrikelnummer liefert und überschreibt die Methode asString, wobei sie die überschriebene Methode verwendet.

- Problematik: Der Mechanismus der Vererbung führt leicht dazu, Spezialisierungen unter dem Aspekt der Wiederverwendung von Implementierungsbestandteilen vorzunehmen, obwohl *logisch* die Klasse, von der man spezialisiert, gar keine Generalisierung der neuen Klasse ist, siehe Pinguin-Beispiel im Kurs auf Seite 101/102
- Eine Subklasse kann auch Methoden der Superklasse löschen, zumindest in Smalltalk, indem sie die Methode so überschreibt, dass diese die Methode shouldNotImplement auf self aufruft, was zu einer Fehlermeldung führt, wenn man versucht, die gelöschte Methode aufzurufen. In anderen Programmiersprachen ist das meist nicht so einfach... das hat Gründe: Das Löschen einer Methode steht natürlich im krassen Widerspruch zu dem Gedanken, dass eine Superklasse Generalisierung ihrer Subklassen ist! Anders gesagt: Wenn der Wunsch aufkommt, in einer Subklasse eine Methode zu löschen, dann ist das ein sicheres Indiz, dass die Superklasse, die man spezialisieren will, gar keine Generalisierung der neuen Klasse ist!

Abstrakte Klassen

Generalisierungen sind eigentlich per definitionem abstrakt: Wenn es für die Abstraktionsebene eines Programms sinnvoll erscheint, Exemplare einer Klasse Student zu haben, ergibt es keinen Sinn, dass es auch direkte Exemplare der Klasse Person gibt: Jede Person ist entweder Student oder Exemplar einer anderen Spezialisierung von Person. Das programmiersprachliche Äquivalent solcher Klassen, zu der es keine direkten Instanzen gibt, sind abstrakte Klassen. Abstrakten Klassen fehlen in der Regel Teile der Verhaltensspezifikation, nämlich solche Teile, welche erst die Subklassen auf ihre jeweils spezielle Weise beitragen können. Dennoch können abstrakte Klassen zumindest spezifizieren, dass eine individuelle Implementierung eines bestimmten Verhaltens von den Subklassen

erwartet wird. In Smalltalk geschieht dies, indem die betreffende Methode in der abstrakten Klasse zwar existiert, jedoch weiter nichts tut, als ihrerseits die Methode implementedBySubclass aufzurufen, was wieder (analog zum Aufruf einer gelöschten Methode) zu einer Fehlermeldung führt.
→ Verweis auf andere Programmiersprachen, etwa Java, welche die Instanziierung abstrakter Klassen von vornherein durch den Compiler überprüfbar verbieten.

Dass eine abstrakte Klasse die Implementierung bestimmter Methoden in die Verantwortlichkeit konkreter Subklassen legt, heißt natürlich nicht, dass die abstrakte Klasse nur solche abstrakten Methoden hätte. Es kann zum einen Verhalten geben, dass für alle Spezialisierungen gleich ist, entsprechende Methoden sind also sinnvollerweise in der abstrakten Klasse ausprogrammiert.

Besonders interessant ist in diesem Fall, dass aus den bereits implementierten Methoden auch die noch "fehlenden" Methoden aufgerufen werden können: Da das Objekt, auf dem die implementierte Methode aufgerufen wurde, ein Exemplar eines konkreten Subtyps sein muss, wird durch einen solchen Aufruf ja gar nicht die "fehlende" Methode aufgerufen, sondern per dynamischer Bindung die eben dieses konkreten Subtyps. Man spricht in diesem Fall von "offener Rekursion": Der Aufruf erfolgt auf self, insofern also gewissermaßen rekursiv, es ist aber an der aufrufenden Stelle unklar, zu welcher Klasse self überhaupt gehört. Diesen Mechanismus kann man verwenden, indem man das Verhalten der spezialisierenden Subklassen so auf Methoden aufteilt, dass wirklich nur noch genau die Teile in Subklassenmethoden liegen, die wirklich das Besondere der jeweiligen Subklasse abbilden. → Beispiel: Brettspiele mit Feldern, Setzen von Stein auf ein Feld ist nur erlaubt, wenn auf einem Nachbarfeld bereits ein Stein liegt. Unterschiedliche Spiele mit unterschiedlichen Feldanordnungen, z.B. Quadrate, Hexagone. Die Berechnung, auf welche Felder bei gegebener Stellung Steine gesetzt werden können, kann fast komplett in einer abstrakten Stellung-Klasse erfolgen. Die konkreten Subklasen liefern nur noch die Nachbarfelder zu einem gegebenen Feld, die entsprechende Methode wird aus den Methoden der abstrakten Klasse verwendet.

Typen

Voraberklärung: Typsystem ist nicht zwingender Bestandteil von OOP, siehe Smalltalk. Smalltalk hat zwar Klassen, aber keine typisierten Variablen.

Allerdings ist es wesentlicher Bestandteil aller Mainstream-OOP-Sprachen und Fragen, die mit dem Typsystem zu tun haben, füllen einen wesentlichen Teil des Kurses. Wieso eigentlich?

Was ist eigentlich ein Typ? In Pascal eine Beschreibung einer Datenstruktur, die gemeinsame Merkmale der „Exemplare“ der Datenstruktur zusammenfasst. In OOP?

Warum Typen? → Wunsch nach statischer Typsicherheit: Zuweisungen und Gültigkeit von Methodenaufrufen können günstigstenfalls bereits durch Compiler überprüft werden, d.h. es ist sichergestellt, dass ein Objekt eine ihm geschickte Nachricht auch verarbeiten kann.

Unflexibles Typsystem, z.B. Pascal: Erweiterung eines Typs nicht möglich.
Sprache ohne typisierte Variablen, z.B. Smalltalk: Laufzeitfehler, wenn Objekt Nachricht nicht verarbeiten kann.

Was will man? Einerseits statische Typsicherheit, andererseits Flexibilität.
Lösung (mit Einschränkungen): Subtyping mit Auseinanderfallen des statischen / dynamischen Typs
→ Dynamische Methodenwahl, dynamisches Binden.

Subtyping

Subtyp-Beziehung: „Ist-ein-Beziehung“ zwischen Typen. Nicht verwechseln mit „Ist-ein-Beziehung“ zwischen Objekt und seiner Klasse! Bezug zwischen beiden Beziehungen: Aus der Beziehung "Hund ist Subtyp von Tier" folgt: "Jedes Exemplar der Klasse Hund ist auch Exemplar der Klasse Tier", aber nicht etwa, dass ein konketes Exemplar der Klasse Hund eine Klasse wäre oder dass die Klasse Hund Exemplar der Klasse Tier wäre. Zur Sicherheit die Beziehung präziser benennen: Beziehung zwischen Typen als "ist ein Subtyp von", Beziehung zwischen Exemplar und seinem Typ als "ist ein Exemplar von".

Substituierbarkeit: Ein Objekt eines Subtyps muss überall stehen können, wo ein Objekt des Supertyps erlaubt ist. → Ein Objekt des Subtyps muss alle Nachrichten verstehen können, die man einem Objekt des Supertyps schicken kann. → Objekte des Subtyps müssen mindestens die Schnittstelle der Objekte des Supertyps haben.

Vererbung, Überschreiben

Subtyp „erbt“ Methoden des Supertyps, kann diese aber durch eigene Methoden ergänzen. Außerdem können Methoden durch neu definierte Methoden „ersetzt“ werden. Letzteres nennt man „Überschreiben“. Achtung: Da ein Objekt des Subtyps auch ein Objekt des Supertyps ist, ist eine überschriebene Methoden durchaus noch vorhanden, sie ist nur im Subtyp durch die überschreibende Methode „verdeckt“.

1. Logische Folgerung aus der Forderung nach Substituierbarkeit:

Beim Überschreiben von Methoden ist nur Kovarianz beim Rückgabetyp und Kontravarianz bei den Parametertypen mit dem LSP vereinbar. Anders gesagt: Eine überschreibende Methode muss alle Parameter „verkraften“, welche die überschriebene Methode verträgt und sie darf keine Typen zurückliefern, die nicht auch die überschriebene Methode hätte liefern können.
Insbesondere Ersteres ist sehr schade, denn eigentlich möchte man zur Modellierung der Realität Kovarianz bzgl. der Parametertypen. Beispiel: Tier hat Methode friss(Nahrung n). Dann darf der Subtyp Kuh diese Methode nach dem oben Gesagten nicht so überschreiben, dass die überschreibende Methode als Parameter den Typ Gras deklariert, denn das wäre Kovarianz bzgl. des Parametertyps und würde dem LSP widersprechen, denn eine Kuh würde eben nicht das Schnitzel „verkraften“, welches der Tierpfleger an alle Tiere verteilt. → Widerspruch zwischen Substituierbarkeitsbedingung und „ist-ein“-Beziehung der Realwelt. Ursache ist letztlich die in obiger Methodendeklaration implizit enthaltene Allquantifizierung: Das Vorhandensein der Methode friss(Nahrung n) in Tier postuliert eben, dass man *jedem* Tier *jede* Nahrung zu fressen geben kann und das ist in Subtypen nicht ohne Verlust der Substituierbarkeit wieder zurückzunehmen.

2. Logische Folgerung aus der Forderung nach Substituierbarkeit:
Forderung nach konformem Verhalten überschreibender Methoden. Eine überschreibende Methode muss sich so verhalten, dass dieses Verhalten mit der Spezifikation der überschriebenen Methode vereinbar ist. → Starke Einschränkung der Flexibilität beim Erweitern eines Typs, Forderung wird in der Praxis oft verletzt, da sie im Ggs. zu den Forderungen bzgl. Ko-/Kontravarianz von den Compilern gängiger Sprachen nicht überprüft werden kann.

Achtung: Java erlaubt *keine* Kontravarianz beim Parametertyp, obwohl diese logisch möglich ist! Grund: Es bedürfte komplizierter Regeln, um zu entscheiden, wann überschrieben und wann überladen wird. Das ist aber kein Verlust: Es lassen sich kaum sinnvolle Beispiele für den Wunsch nach dem Überschreiben einer Methode mit kontravariantem Parametertyp finden.

<u>Unterschied Subtyping vs. Subclassing vs. Vererbung vs. Delegation</u>

Vier Konzepte, die konzeptionell unabhängig voneinander sind!

- Subclassing: Von einer Klasse wird eine neue, speziellere Klasse abgeleitet. In Java hat man bei Subclassing immer auch Subtyping und Vererbung.
- Vererbung: Implementierungsbestandteile eines Typs sind automatisch in einem abgeleiteten Typ verfügbar. Vererbung zwischen Supertyp und Subtyp ist in vielen Sprachen der Weg, um sicherzustellen, dass ein Subtyp alle Nachrichten versteht, die der Supertyp versteht.
- Subtyping: siehe oben. In Java ermöglichen Interfaces Subtyping ohne Vererbung
- Delegation: Wiederverwendung bestehenden Codes, indem die zu erledigende Aufgabe an ein anderes Objekt – das sog. Delegate – „durchgereicht“ wird. Delegation ist ein Weg, bestehenden Code wiederzuverwenden, ohne eine Subtypbeziehung zu begründen. Man kann Vererbung konzeptionell auch als einen speziellen Fall von Delegation (vom Subtyp an den Supertyp) betrachten

Polymorphie

Ganz allgemein: Polymorphie ermöglicht, Konstrukte so zu programmieren, dass sie mit verschiedenen Typen arbeiten können. Verschiedene Formen von Polymorphie:

Subtyp-Polymorphie: Ergibt sich direkt aus dem LSP. Beispiel: Ein Container, dessen Elementtyp Object ist, kann beliebige Objekte aufnehmen. Problem: Bzgl. Der Elemente ist später als Typ nur noch Object bekannt. Eine Methode, die ein Element liefert, hat als Rückgabetyp Object. Wenn man die Elemente wieder mit dem eigentlichen Typ verwenden will, muss man dies dem Compiler zusichern (Cast) → Verlust der Typsicherheit.

Parametrische Polymorphie: Annotation mit Typparametern. Nur Elemente des betreffenden Typs sind erlaubt (bei gleichzeitiger Subtyppolymorphie natürlich auch Elemente von Subtypen). Bei der Erzeugung eines Exemplars wird der Typ festgelegt, d.h. Verwender können z.B. einen Container nur für String-Exemplare erzeugen. Für *welchen* Typ der Verwender den BinaryTree erzeugt, ist ihm aber völlig freigestellt. Problem: Der Ersteller des BinaryTrees kann keine Annahmen bzgl. des späteren Elementtyps treffen. Z.B. kann nicht gewährleistet werden, dass die Elemente (zum Sortieren) Subtyp von Comparable (vergleichbar) sind.

Beschränkte Parametrische Polymorphie: Wie Parametrische Polymorphie, nur schränkt der Ersteller die Möglichkeiten des Exemplar-Erzeugers im Voraus ein, indem er eine Schranke (oder auch mehrere) für den Typparameter festlegt. Damit kann z.B. der Ersteller eines Containers festlegen, dass der Exemplar-Erzeuger als Typparameter nur Subtypen von Comparable übergeben kann und diese Zusatzinformation kann er in seinem Code verwenden.

Rekursiv beschränkte parametrische Polymorphie: Bei der Definition eines Typs wird dieser Typ selbst direkt oder indirekt als Typparameter verwendet. Beispiel: Um auszudrücken, dass ein String vergleichbar zu Strings ist, schreibt man: class String implements Comparable<String>

Ad-hoc-Polymorphie (Überladen): Methoden gleichen Namens aber mit unterschiedlichen Parametertypen werden als „überladen“ bezeichnet. Überladung wird in Java durch den Compiler ausschließlich anhand der *Deklarationstypen* der Parameter aufgelöst. Das geschieht, indem aus der Liste der potentiell passenden (s.u. bei „Compiler vs. Laufzeitumgebung“) Methoden die speziellste ausgewählt wird, indem jede Methode „gestrichen“ wird, zu der es eine speziellere gibt. Bleibt dabei mehr als eine Methode übrig, ist der Aufruf unentscheidbar (ambigeous) → Compilerfehler. Überladen ist keine „echte“ Polymorphie, weil hier ja gar nicht derselbe Code für verschiedene Konstrukte verwendet wird, sondern es primär darum geht, für Methoden, die eine prinzipiell gleiche Aufgabe anhand verschiedener Parameter unterschiedlich erledigen, den gleichen Namen vergeben zu können. Polymorph ist hier also höchstens der Name als solcher. Überladung bitte *nie* mit Überschreiben verwechseln!

Codebeispiele zu Polymorphie siehe Studientags-Workspace und das PDF Hinweise_zu_den_Codebeispielen.pdf

Anmerkungen zum Typsystem Javas

Javas Typsystem bietet (zum Preis diverser Einschränkungen) statische Typsicherheit, mit folgenden Ausnahmen:

- Casts. Ein expliziter Downcast oder Crosscast hebelt zwangsläufig die statische Typprüfung aus. Es erfolgt aber eine Prüfung zur Laufzeit, ggf. wird eine ClassCastException ausgelöst.
- Eine eventuelle Subtypbeziehung zwischen den Basistypen eines Arraytyps überträgt sich auf die Arraytypen selbst. Wenn also gilt `Student` SUB→ `Person`, dann gilt auch `Student[]` SUB→ `Person[]`. Das ist prinzipiell durchaus sinnvoll, denn es erlaubt sinnvolle polymorphe Konstrukte, die z.B. mit allen Arrays arbeiten können, deren Basistyp Subtyp von Person ist. Aber:

  ```
  Student[] s = new Student[2];
  Person[] p = s;
  p[0] = new Person();
  ```

  Jeden Zeile für sich sieht unproblematisch aus, aber am Ende hätten wir in einem Student[] ein Person-Exemplar, was ein Typfehler wäre. Deshalb erfolgt bei der Zuweisung in der dritten Zeile eine Typüberprüfung zur Laufzeit, bei der sich herausstellt, dass der von p referenzierte Array ein `Student[]` ist. → ArrayStoreException. Man ist hier also einen (durchaus begründeten) Kompromiss zuungusten der statischen Typsicherheit eingegangen. Davon ist man bei generischen Typen dann allerdings wieder abgekommen... es gilt also z.B. *nicht* `List<Student>` SUB→ `List<Person>`
- Die Verwendung eigentlich generisch deklarierter Typen in Form sogenannter "Raw Types" hebelt die statische Typsicherheit aus, im Extremfall sogar die Typprüfung zur Laufzeit. Das kann zu schwer auffindbaren Fehlern führen, ist aber leicht zu vermeiden, indem man eben kein Raw Types verwendet. Die Möglichkeit, Raw Types überhaupt zu verwenden, hat "historische Gründe": Abwärtskompatibilität zu Code, der vor Java 1.5 geschrieben wurde, Einbindung entsprechender Alt-Bibliotheken.

Möglichkeiten und Restriktionen von WildcardTypen (am Beispiel Javas):

Motivation: Analog zu dem o.g. Sachverhalt bei Arrays möchte man z.B. für generische Collections eine Methode sort definieren können, der man eine beliebige Liste zum Sortieren übergeben kann, also etwa ein Exemplar von ArrayList<Integer> oder ArrayList<String>. Naheliegende Idee: Parametertyp ist z.B. ArrayList<Comparable>. Das geht aber nicht, denn obwohl Integer uns String Subtypen von Comparable sind, sind eben ArrayList<Integer> und ArrayList<String> *nicht* Subtypen von ArrayList<Comparable>.

Lösung: Sogenannte Wildcard als Platzhalter für einen beliebigen Typ. Beispiel: List<?>. Dieser Typ ist definitionsgemäß Supertyp aller Instanziierungen von List<T>, also etwa auch ArrayList<Integer> und ArrayList<String>.

Das reicht aber noch nicht... denn um in unserer Sortiermethode auf den Elementen der übergebenen Liste deren Methode compareTo() aufrufen zu können, benötigen wir eine Garantie, dass der Typ, mit dem die konkrete Liste parametrisiert wurde, Comparable oder ein Subtyp von Comparable ist. M.a.W.: Wir brauchen einen spezielleren Typ als den "aller Instanziierungen von List<T>". Lö-

sung: Beschränkte Wildcards. Der Typ List<? extends Comparable> leistet, was wir wollen. Er ist Supertyp von ArrayList<Integer> und ArrayList<String>, nicht aber z.B. von ArrayList<Object>, denn Object ist ja kein Subtyp von Comparable.

Die Beschränkung des Parametertyps "nach oben" impliziert die Frage, ob es auch eine Möglichkeit gibt, den Parametertyp "nach unten" zu beschränken und wozu das gut sein könnte. Die Möglichkeit existiert in der Tat, das dem "extends" von oben entsprechende Schlüsselwort ist "super".

Codebeispiele zum "Schreiben" in bzw. dem "Lesen" aus einer Liste, die man nur über einen Wildcard-Typ kennt siehe Studientags-Workspace, Package bsp_17_wildcards und Package bsp_18_wildcards

Fragile Base Class Problem

Eigentlich nicht ganz korrekte Benennung: Fragil (zerbrechlich) ist nicht die Basisklasse (Superklasse), sondern von ihr abgeleitete Klassen "zerbrechen" aufgrund von scheinbar harmlosen Änderungen in der Superklasse, nämlich solchen, die nicht einmal die Klassenschnittstelle ändern.

Einfaches Beispiel siehe Studientags-Workspace, Package bsp_19_fragile_base_class

Programmiersprachen – wesentliche Merkmale / Besonderheiten (unvollständig!)

Smalltalk

- alles ist ein Objekt
- Sprache ohne Typsystem, Typfehler werden erst „erkannt“, wenn ein Objekt eine an es gesendete Nachricht nicht bearbeiten kann
- klassenbasiert
- hochgradig dynamisch, Laufzeitsystem und IDE sind Bestandteile *eines* integrierten Systems
- Elemente aus der funktionalen Programmierung: Blockclosures, Continuation
- automatische Speicherbereinigung (Garbage Collection)
- Objekte sind nur über Referenzen zugreifbar, Parameterübergabe erfolgt durch Call by value

Java

- Sprache mit statischer und dynamischer Typprüfung, bieter weitgehende statische Typsicherheit
- unterscheidet zwischen Primitivtypen und Objekten; dieser Unterschied ist seit Java 1.5 durch Autoboxing weniger sichtbar. Klassen, Methoden usw. sind keine echten Objekte, auch wenn inzwischen mit der Reflection-API eine Möglichkeit existiert, sie begrenzt zum Gegenstand der Programmierung zu machen.
- Subtyppolymorphie, parametrische Polymorphie, beschränkt parametrische Polymorphie
- keine Mehrfachvererbung, aber mehrfaches Subtyping
- Interface-Typen ermöglichen Subtyping ohne Vererbung, das Modellieren von Rollen/Sichten auf Objekte und das Programmieren „gegen vereinbarte Schnittstellen“
- automatische Speicherbereinigung (Garbage Collection)
- Objekte sind nur über Referenzen zugreifbar, Parameterübergabe erfolgt durch Call by value
- Compiler erzeugt Zwischencode (Bytecode), Ausführung durch Laufzeitumgebung (Virtual Machine), Optimierung durch Just-in-time-Compiler

C++

- klassenbasiert
-
- Speicherverwaltung liegt in der Verantwortung des Programmierers

- bietet Mehrfachvererbung
- Explizite Pointer, Parameterübergabe kann per Call by value oder per Call by reference erfolgen

C#

Eiffel

Compiler vs. Laufzeitumgebung bei Java:

Der Compiler kennt nur die Deklarationstypen von Variablen/Ausdrücken und nur diese berücksichtigt er bei der Auflösung von Methodenaufrufen. Ein Objekt welchen Typs eine Referenz zur Laufzeit referenzieren wird, kann der Compiler nicht wissen, also spielt für alles, was der Compiler tut, auch der Typs dieses potentiellen Objekts keine Rolle. Das wirkt sich insbesondere aus bei:

- der Entscheidung, welche Methoden überhaupt über einen Ausdruck / eine Variable aufrufbar sind. Das sind immer nur Methoden, die für den Deklarationstyp des Ausdrucks definiert sind, und von diesen genau die, bei denen die Deklarationstypen der aktuellen Parameter jeweils Subtypen der Deklarationstypen der formalen Parameter sind.
- bei der Auflösung von Überladung. Hier spielen nur die Deklarationstypen der Beteiligten eine Rolle. Diese Aussage gilt für Java; es gibt Sprachen, die sog. Multimethoden bieten; dabei findet die dynamische Methodenwahl auch anhand der Laufzeittypen der Parameter statt, während sie in Java nur anhand des Laufzeittyps des Nachrichtenempfängers (des sog. „impliziten Parameters“) erfolgt.

Als Ergebnis der Analyse eines Methodenaufrufs schreibt der Compiler den Aufruf einer der für Deklarationstyp definierten Methoden in den Bytecode. Erst die Laufzeitumgebung schaut dann nach, welchen Typ der tatsächliche Nachrichtenempfänger hat. Sie sucht dann nach einer Implementierung genau der Methode, die im Bytecode steht, wobei sie die Suche in der Klasse des Nachrichtenempfängers beginnt, und sich in der Vererbungshierarchie nach „oben“ hangelt, bis sie eine Implementierung findet. Diese Methode wird dann ausgeführt. Die Typen der Parameter spielen zur Laufzeit also keine Rolle mehr.
Anders gesagt: Wenn wir zwei Typen Sub und Super haben und eine Methode mit einer bestimmten Parameterkombination ist in Sub definiert, dann kann sie überhaupt nur zur Ausführung gelangen, wenn:

- der Aufruf auf einem Ausdruck erfolgt, der eben Sub als Deklarationstyp hat,
  oder
- wenn die Methode eine Methode des Supertyps überschreibt.

Näheres hierzu findet sich in dem PDF 1618-FAQ.pdf

Exceptions

Klassische prozedurale Programmierung: Beim Auftreten eines Fehlers (Dereferenzierung von nil, Teilen durch 0, Dateioperationen auf nicht verfügbarer Datei usw.) terminiert das Programm mit einer mehr oder weniger aussagekräftigen Fehlermeldung. Will man dies vermeiden, muss man den Fehler durch entsprechende Abfragen im Vorfeld vermeiden und ggf. einen alternativen Ablauf programmieren. Nachteile: Fehlerbehandlung muss entweder direkt vor Ort erfolgen oder erfordert Sprünge → Strukturiertes Programmieren wird erschwert. Fehlerbehandlung ist unflexibel. Der Compiler überprüft nicht, ob mögliche Fehler behandelt werden.

Exception-Mechanismus:

- ermöglicht, eine Problemsituation dort zu behandeln, wo diese Behandlung sinnvoll ist. Beispiel: Meldung an den Benutzer aus einer I/O-Bibliotheksmethode heraus ist *nicht* sinnvoll: Je nach Einsatzumgebung müsste die Meldung auf völlig unterschiedliche Weise erfolgen (GUI, Konsolenausgabe, Mail an Admin).
- ermöglicht, für eine Methode das potentielle Auftreten einer Ausnahmesituation zu deklarieren, so dass der Compiler überprüfen kann, ob der Verwender der Methode dies berücksichtigt.
- ermöglicht dem Programmierer, gezielt Informationen über die Umstände, welche die Ausnahme ausgelöst haben, an die Stelle zu übermitteln, welche die Ausnahme behandelt.

Java: Drei Arten von "Throwables": Error, Exception, RuntimeException.

- Error (Subklassen von Error): Steht für Fehler, mit denen der Programmierer nicht rechnen muss bzw. bei denen normalerweise zur Laufzeit keine Eingriffsmöglichkeit besteht: Bugs in der VM, zu wenig Speicher, usw. Ein Abfangen eines Errors ist normalerweise nicht sinnvoll.
- Checked Exceptions (Subklassen von Exception, die nicht Subklasse von RuntimeException sind): Die "normale Ausnahme": Eine besondere Situation, die durch den Programmierer vorhersehbar auftreten kann und einen alternativen Programmablauf erfordert. Die Behandlung von Checked Exceptions wird durch den Compiler gemäß der Regel "Catch or Declare" überprüft.
- Unchecked Exceptions (Subklassen von RuntimeException): Eine Ausnahme, die typischerweise durch einen Fehler des Programmieres verursacht wird. Eine Behandlung von Unchecked Exceptions ist normalerweise *nicht* sinnvoll, das sie prinzipiell *immer* auftreten können: Bei jeder Division könnte der Divisor 0 sein, bei jeder Dereferenzierung besteht die Möglichkeit, dass man die leere Referenz vor sich hat (ArithmeticException und NullPointerException sind typische Unchecked Exceptions).

Erläuterung "Catch or Declare":

- Mit Hilfe des Schlüsselworts "throws" deklariert eine Methode (oder ein Konstruktor), dass sie unter Umständen eine bestimmte Exception auslösen ("werfen") *kann*.
- Stößt der Compiler auf einen Aufruf einer solchen Methode, und bei der deklarierten Exception handelt es sich um eine Checked Exception, prüft er, ob dieser Aufruf selbst in einen try-Block eingebettet ist, zu dem ein für die betreffende Ausnhame passender catch-Block existiert *oder*

ob die Methode, in welcher der Aufruf steht, selbst deklariert, diese Exception (oder einen ihrer Supertypen) auslösen zu können. Ist beides nicht gegeben → Compilerfehler.
- Das tatsächliche Auslösen ("Werfen") einer Exception geschieht durch das Schlüsselwort "throw", gefolgt von einem Ausdruck, der eine Referenz auf ein Exception-Exemplar liefert, typischerweise der Konstruktor einer Exception-Klasse, also etwa: `throw new MyException("Text");`

Vorgänge zur Laufzeit:

Wird eine Exception ausgelöst, wird geschaut, ob sich die throw-Anweisung innerhalb eines try/catch-Blocks befindet, der über mindestens eine passende catch-Klausel verfügt. Ist dies der Fall, wird die Ausführung in der ersten passenden catch-Klausel fortgesetzt und, wenn diese regulär beendet wurde, hinter der schließenden Klammer des try/catch-Blocks weitergemacht.

Steht die throw-Anweisung nicht in einem try/catch-Block oder existiert keine passende catch-Klausel, terminiert die umgebende Methode abrupt, d.h. der Aufruf dieser Methode kehrt mit der ausgelösten Exception an die Aufrufstelle zurück. Dort wird dann der Aufruf der Methode so behandelt, wie oben für die throw-Anweisung beschrieben: Es wird geschaut, ob sich der Methodenaufruf innerhalb eines try/catch-Blocks befindet, der über mindestens eine passende catch-Klausel verfügt. Ist dies der Fall, wird die Ausführung in der ersten passenden Catch-Klausel fortgesetzt und, wenn diese regulär beendet wurde, hinter der schließenden Klammer des try/catch-Blocks weitergemacht.

Steht der Methodenaufruf nicht in einem try/catch-Block oder existiert keine passende catch-Klausel, terminiert die umgebende Methode abrupt... dieses Spiel setzt sich fort, bis entweder ein passender catch-Block gefunden wird, oder die äußerste Methode (das ist die main-Methode oder die run-Methode des betreffenden Threads) abrupt terminiert. In diesem Fall wird normalerwiese ein sog. Stacktrace auf die Konsole ausgegeben.

Sonderfall finally: Gehört zu einem try-Block ein finally-Block, wird dieser *immer* beim Verlassen des zugehörigen try-Blocks ausgeführt, also sowohl, wenn der try-Block ganz normal abgearbeitet wurde, als auch, nachdem eine aufgetretene Exception in einem zugehörigen catch-Block behandelt wurde und auch, wenn kein passender catch-Block gefunden wurde und der try-Block (und damit die umgebende Methode) abrupt terminiert.