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."

Die Existenz einer klar definierten Schnittstelle 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.
- Objekte können nach ihrer Schnittstelle klassifiziert werden: Objekte mit gleicher Schnittstelle bilden eine Klasse, wobei dieser Begriff hier zunächst einmal nicht im Sinne einer Klasse einer Programmiersprache zu verstehen ist, auch wenn es natürlich einen Zusammenhang gibt.
- Naheliegende Umsetzung: Schnittstellenbeschreibung nicht pro Objekt, sondern pro Klasse.

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 Nachricht 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.

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. Variablen können zu verschiedenen Zeitpunkten Referenzen auf verschiedene Objekte zugewiesen werden, 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 Nachrichtenversands 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, VM, Runtime Engine usw.). Normalfall bei objektorientierter Programmierung.

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 1618. 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 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).

Liskovsches Substitutionsprinzip: 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“.

1. Logische Folgerung aus dem LSP:

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 LSP 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 Subtypbeziehung laut LSP und „ist ein“-Beziehung der Realwelt.

2. Logische Folgerung aus dem LSP:

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 und, wichtiger: Überschreiben mit kontravariantem Parametertyp ist eh‘ unbrauchbar, weil in der Praxis speziellere Typen immer auch speziellere Eingangsparameter ihrer Methoden haben und eben nicht allgemeinere (siehe oben: Kuh frisst Gras, was spezielle Nahrung ist).

Unterschied Subtyping vs. Subclassing vs. Vererbung vs. Delegation

Vier Konzepte, die konzeptionell erst einmal 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. Ein typisches Beispiel findet man im Kurs in der Ad-hoc-Aufgabe 7 zur KE 1. Dort werden Aufgaben an eine ArrayList delegiert. 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 Container erzeugt, ist ihm aber völlig freigestellt. Problem: Der Ersteller des Containers 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.

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 ähnliche Aufgabe anhand verschiedener Parameter unterschiedlich erledigen, den gleichen Namen vergeben zu können. Was hier also polymorph verwendet wird, ist eigentlich nur der Methodenname.

Überladung.bitte *nie* mit Überschreiben verwechseln!

Codebeispiele zu Polymorphie siehe Studientags-Workspace und das PDF Hinweise_zu_den_Codebeispielen.pdf, außerdem gibt es Beispiele in der 1618-FAQ und in diversen früheren Klausuren.

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 Methode mit einer bestimmten Methodensignatur in den Bytecode. Erst die Laufzeitumgebung schaut dann nach, welchen Typ der tatsächliche Nachrichtenempfänger hat. Sie sucht dann in diesem Typ nach einer Methode, mit genau der Signatur, die der Compiler gewählt hat., 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 Aufrufparameter 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.

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 Java ist die Instanziierung abstrakter Klassen durch den Compiler überprüfbar verboten.

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 this, insofern also gewissermaßen rekursiv, es ist aber an der aufrufenden Stelle unklar, zu welcher Klasse this 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.

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.

Dagegen 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 vor- hersehbar 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.