INFORMATIK II
am D-BAUG

Kurzzusammenfassung zur Vorlesung 252-0846-00
Frühjahrssemster 2014, ETH Zürich

Felix Friedrich

Chapter 1: Introduction

We saw an overview over the planned topics of the course: object oriented programming, data structures and algorithms, data bases. I pointed out that the focus of the course is on *solving problems* and not primarily on learning a programming language. This is underpinned by the discussion of *case studies* per learned concepts in the course.

We shortly recapitulated the concept of computers and programming.

We treated the programming language for this course, Java, as an imperative language first by comparing it with Pascal. We learned how basic expressions and statements are translated from Pascal to Java and already looked at the major differences: existence of classes (constituting reference typed) and methods as opposed to (value typed) records and procedures. We learned that Java does not provide reference parameters. We learned what the simplest Java program looks like ("hello world").

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

Case study: Ancient Egyptian Multiplication

We learned the algorithm of ancient Egyptian multiplication: a method to multiply two numbers by only using multiplications and divisions by 2 and additions. Starting from a formal *recursive* formulation of the algorithm, we discussed an implementation in Java. We gave an inductive proof of the correctness of the algorithm. This was followed by a discussion about the limits of applicability of such theoretical result to programs.

We introduced the concept of *exception handling* and learned that it breaks the normal control flow in order to react on an exceptional situation, such as the overflow of a stack when the algorithm did not terminate. Recall that a program can either correctly terminate, return an incorrect result (including ungraceful termination) or not terminate at all.

```
try {
// statements
}
catch (errtype1) {
// handling of this error type
}
```

By elementary transformations of the code we formulated the algorithm in a *tail-recursive* and an *iterative* form. We identified *invariants*, an important tool for reasoning about the correctness of code.

Chapter 2. Java

Compilation units in Java consist of *classes* that can be contained in *packages*. We discussed the role of *import* with respect to *name spaces* and *qualified identifiers* in Java. *Static* methods play the role of procedures in Pascal. A self-contained program provides a class containing a method of the form `public static void main(String args[])`

Java is strongly typed and therefore assignments between symbols of different type require type conversions. When converting to a type with larger domain an *implicit* conversion takes place. The compiler detects static incompatibilities while the runtime can detect dynamic incompatibilities.

Arrays are dynamic objects in Java that can be (re-)allocated during runtime. As a consequence of reference semantics in Java, an assignment of variables of type array does not imply a data copy but only a copy of the *reference* to the data.

```
int [] x = new int[10];
int [] y;
y = x;      // copies the reference, not the array data
y[3] = 10; // modifies y and consequently also x !
```

Array bounds are checked at runtime. As an example of arrays we learned how to access the arguments args of a java program. Input, output and error Streams can be used in order to write to or read from the console. Using the symbols < , > and |, in- and output of a program can be redirected to files or *piped* between java programs.

Case Study: Random Surfer, Page Rank Algorithm

A random surfer starts at an arbitrary page on the internet and continues iteratively choosing outgoing links with equal probabilities. We modeled the behavior of the random surfer as a Markov Chain with transition matrix $\mathbf{P} := (P_{ij})_{0\leq i,j\leq n}$. Entries P_{ij} stand for the probabilities to continue on page j when having started on page i.

Most importantly, we developed a way to *simulate* a finite random variable $V \in \{0, ..., n-1\}$ with given distribution $p_i = \mathbb{P}(V = i)$.

```
public static int Simulate(double[] p)
{
    int res=0;
    double r = Math.random();
    double sum = 0.0;
    for (int j = 0; j < p.length; j++) {
        sum += p[j];
        if (r < sum) {res = j; break;}
    }
    return res;
}
```

This procedure was iteratively applied taking respective rows of the transition matrix $\mathbf{P}$ in order to simulate the behavior of the random surfer. We computed the *page rank* as visiting frequency for each page.

Chapter 3. Classes

Classes in Java contain data *and* code. Classes have reference semantics, i.e. they have to be allocated with new. Deallocation is not necessary as Java comes with a Garbage collector.

We learned about the syntax and semantics of the new statement. A class can provide several constructors with different *signature*. A constructor is called when new(...) is executed according to its parameters, following the principle of method *overloading*.

```
class Rational {..
    public Rational(int num, int denom){ ... }
    public Rational(){ ... } ...
}
```

Encapsulation is an important concept which helps to give guarantees regarding invariants. By hiding implementation details behind a defined interface it permits to provide a sufficient level of abstraction.

Methods have access to the data (variables) of their containing class via the implicit this parameter. If this is not specified in a reference, it is implicitly added provided that the symbol refers to the class variables.

Passing a variable of class type as parameter means passing a *reference* to an object.

Case Study: Online Statistics

Wanted: object providing values such as mean or variance without costly computation.

The *circular buffer* is a concept to store a limited amount of data. Using this concept mean and variance can be computed in linear time. Using the concept of a *dynamic (growing) array*, the amount of stored data can be derestricted.

For online computation of mean and variance, the provisional means algorithm is much better. It updates the mean according to the formula $\mu_{n+1} = \mu_n + \frac{x_{n+1}-\mu_n}{n+1}$. The median is a selection problem that requires a more complicated treatment.

The Statistics class is a good example of how *getters* and *setters* are used in Object Oriented Programming in order to hide implementation details.

```
public class Statistics {
    int n = 0;    double mean = 0;    double ssq = 0;
    public void Put(double value){
        n++; double oldMean = mean;
        mean = oldMean + (value - oldMean) / n;
        ssq = ssq + (ssq - oldMean) * (value - mean);
    }
    public double Mean(){ return mean; }
    public double Variance() { if (n==0) return 0; return ssq}
}
```

Chapter 4. Complexity

Algorithms use resources such as computing time, memory and energy. We consider problems with problem size n. When computing time consumption of an algorithm, constant sizes should be abstracted away.

We often distinguish scenarios: "best case", "average case" and "worst case". Moreover, often the algorithm behavior is considered for the case $n \to \infty$. For small n an algorithm with worse asymptotic complexity can be actually better.

The big-O notation helps to prescind from constants that do not contribute to the "nature" of the problem or its solution. Formally we say an algorithm is $O(g)$ (of Order g) if

$$\exists c > 0, n_0 \in \mathbb{N} : f(n) \leq c \cdot g(n) \, \forall n \geq n_0$$

We categorized into

constant	$O(1)$
logarithmic	$O(\log n)$
linear	$O(n)$
quadratic	$O(n^2)$
polynomial	$O(n^k)$
exponential	$O(c^n)$.

and realized that the growth for an algorithm with exponential complexity is extreme and renders it in most cases completely unrealistic.

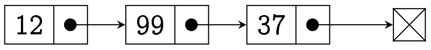
We made the observation that if we can solve a problem of size N within a certain time, then with a new, ten times faster, machine we can solve within the same time depending on the complexity of the algorithm:

$f(n)$	old size $\to$ new size
$O(\log n)$	$N \to N^{10}$
$O(n)$	$N \to 10 \cdot N$
$O(n \log n)$	$N \to (nearly) 10 \cdot N$
$O(n^2)$	$N \to \sqrt{10} \cdot N$
$O(n^k)$	$N \to N^{1/3} \cdot N$
$O(c^n)$	$N \to N + \log_2 10$

We computed the complexity of the mergesort algorithm in a constructive and a recursive way. It is $n + n \log n$

Chapter 5. Dynamic Data Structures I

We motivated dynamic data structures with the observation that for arrays it is computationally expensive to insert or remove elements "in the middle". *Linked lists* provide a solution where elements are stored anywhere in memory (and not consecutively as arrays do it). In addition to the *key* (or value), each list element also stores a pointer to its successor:



For the implementation of a singly linked list, we introduced a dynamic data structure

```
class Node
{
    double value;
    Node next;
    Node (double v, Node nxt){
        value=v; next = nxt;
    }
}
```

and used it in order to implement *stack* and *queue*. We learned that *insert and remove* operations require a careful treatment of *special cases* such as "empty list". Moreover, we learned how to use a *"running pointer"* in order to traverse a list. Sorted insertion requires keeping a reference to the previous element when searching for the insertion position.

Case Study: Point-In-Polygon Algorithm

The Jordan curve theorem implies: in order to identify if a point p is in the inner of a polygon, it is sufficient to count intersections of the polygon with an arbitrary half-line l starting in p. The main difficulty with implementations arise at the special cases where l cuts the polygon on its vertices.

We implemented a polygon as a singly linked list of vertices. The PointInPolygon algorithm was implemented firstly such that it operated on horizontal lines taking into account the mentioned special cases. It used floating point arithmetics.

Motivated by potential ambiguities, lacking computing efficiency and by the requirement to draw the polygon, we discussed line drawing on a pixel grid as an interesting *discretization* task. The *Bresenham algorithm* can be employed to draw an arbitrary discretized line on a pixel grid. It works without floating point arithmetic. The Bresenham algorithm keeps the absolute value of $err = dx(d - y_0) - dy(x - x_0)$ small when advancing in horizontal or vertical direction.

The last task of this case study was drawing and filling the polygon. In order to achieve this, the polygon is represented as an array of linked lists containing intersection coordinates. The lists are filled with the Bresenham algorithm.

Chapter 6. Object Oriented Programming

Motivation: build a graphics library for drawing geometric figures. A solution with the procedural approach was sketched and problems were identified: waste of memory resources, administrative overhead, hindered "non-invasive" code extension.

One key to the solution of such problems is the concept of *inheritance*: common properties of figures can stay in a base class ("generalisation") and distinguishing properties can be expressed in the inheriting classes ("specialisation"). Inheritance implies possibility of code reuse operating on common properties. We introduced the notions *base class*, *inheriting class*.

The compatibility rules state that an extended object can be used whenever a base object is required. During runtime an object can be of extended type relative to its static type. Therefore we introduced type guards and type checks.

```
Figure figure = new Rectangle(0,0,10,10);
Rectangle r = (Rectangle)figure;

if (figure instanceof Rectangle)
    System.out.println("figure is rectangle");
else
    System.out.println("figure is not rectangle");
```

In order to take the dynamic type of an object into account, the concept of *polymorphism* was introduced. When a method with same type and signature as in the base class is defined in the inheriting class then the method to be executed is chosen *at runtime*. Object Orientation requires encapsulation, inheritance and polymorphism.

Case Study: Numerical Integration

Objective was the development of a generic software framework for numerical integration of an arbitrary real valued function. We introduced abstract classes

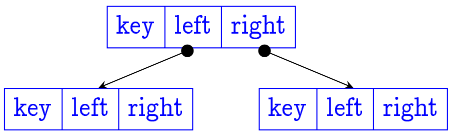
```
public abstract class Function {
    public abstract double Evaluate(double x);
}

public abstract class Integrator {
    int n;
    public void SetNumberPieces(int pieces) {
        n = pieces;
    }
    public abstract double Integrate(Function f, double x0, double x1);
}
```

and experimented with different functions and integrators by specializing. In the course we considered the parabola $f(x) = x^2$ and the density of the normal distribution and applied *Rectangle rule*, *Trapezoidal rule*, *Simpson rule* and a *Monte Carlo* integrator. We derived Simpsons's rule and experimentally verified the error terms of $O(\Delta^3)$ for Rectangle/Trapezoidal rule and and $O(\Delta^5)$ for Simpson Rule.

Chapter 7: Dynamic Data Structures II

Trees are a natural generalization of lists: nodes have more than one successor. A *binary search tree* is a tree of order two with $\text{key}_{x.\text{left}} < \text{key}_x < \text{key}_{x.\text{right}}$ for each node x

```
public class SearchNode {
    int key;
    SearchNode left;
    SearchNode right;
    ...
}
```



Insertion in a binary search tree requires *tree traversal* according to the key order until an empty child node is found. For node removal several cases have to be considered. If the node has two non-empty children then it has to be replaced by a *symmetric successor*, e.g. the leftmost node in its right subtree.

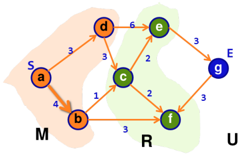
Search trees can *degenerate* to linked lists which implies a worst case complexity $O(n)$. To guarantee $O(\log n)$ in worst case, update operations require additional balancing. *AVL Trees* are trees with height difference between left and right subtrees bounded by 1. Update operations become slightly more complex. At the very heart of balancing are so called *rotations*, code that shifts a node from one subtree to the other.

A *Min-Heap* is a binary tree with the Min-Heap property: the key of a child node is always greater than the key of the parent node. A heap is thus a data structure for fast retreival of the minimum of a data set. The heap data structure can be easily stored in an array with indices $parent(i) = \lfloor (i-1)/2 \rfloor$ and $children(i) = \{2i+1, 2i+2\}$. In order to insert an element in a heap, the element is inserted at the first free position and the heap property is ensured by "raising" the element to its proper position. In order to retreive and delete the minimum element, the root is replaced by the last node and it is "lowered" until the heap property is reasserted.

The median of a data set can be kept up to date with a worst case update complexity $O(\log n)$ by employing a min- and a max-heap around the median.

Case Study: Dijkstra's Shortest Path

Given a directed graph provided with positive weights at the edges, objective is finding a path with lowest accumulated weights leading from starting point S to end point E.



In order to formulate the iterative algorithm of Dijkstra, we considered three sets of nodes. M: nodes that are part of a shortest path, initially $M = \{S\}$. R: all nodes not in M that can be reached via one edge from M and U: all other remaining nodes.

At each update step a node n from R is chosen with minimal path length amongst all nodes in R. Node n is added to M. Then the neighbours of n are added to R and all path lengths of elements in R are updated. In the implementation we used a Min-Heap to store R in order to quickly identify the minimum.

Chapter 8. Tables and Hashing

Motivated by the inefficiency of the "DecreaseKey" operation in our implementation of Dijkstra's algorithm and by the general need to store data sets in a table, the objective was to find a method to store data sets such that an element can be quickly found by key. Data structures considered so far are of limited applicability:

	Search by key	**Update (Insert, Delete)**
array	$O(n)$ (unsorted) $O(\log n)$ (sorted) $O(1)$ (if key = index)	$O(n)$
linked list	$O(n)$	$O(1)$ (unsorted) $O(n)$ (sorted)
search tree	$O(\log n)$ (on average) $O(n)$ (worst case)	$O(\log n)$ (on average) $O(n)$ (worst case)
avl tree	$O(\log n)$ (worst case)	$O(\log n)$ (worst case)

We found that even in the case where the key is an integer, indexing by key in an array is only possible in exceptional cases where the domain of possible key values is very limited.

A *hash function* is a mapping from the set of possible key values to the set of possible indices in an array. A *hash table* is a data structure where references to the data are stored in an array together with a hash function that delivers array indices from key values.

A very simple and often sufficient hash function from the set of integers to the domain of an array with length n is the modulus function

$$h : \mathbb{N} \rightarrow \{0, \dots, n-1\}, k \mapsto k \bmod n.$$

Collisions occur for a hash function h when $h(k_1) = h(k_2)$ for two distinct used key values k_1, k_2. We discussed two possible ways to deal with collisions:

1. Adopt an array of *linked lists* such that each index i refers to a list of all data sets j with $h(key_j) = i$.

2. *Open addressing:* if entry i is occupied with a data set of differing key value a next possible entry is chosen according to a probing procedure. Linear probing: choose next element in array adopting wrap-around semantics. For element deletion, a special symbol is provided in order to not break the lookup algorithm.

A good hash function for strings was identified to be the weighted sum of unicode values

$$h(s) = s_0 \cdot b + s_1 \cdot b^2 + s_2 \cdot b^3 + ... + s_l \cdot b^l$$

for some integer constant $c > 0$ followed by a modulus operation to restrict to the array domain:

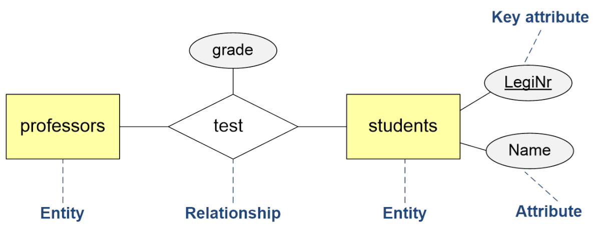
$$index(s) = h(s) \bmod n.$$

Dynamically growing hash tables can be adopted in order to limit the occupancy rate and thus the *collision probability* in the hash table (analogy to birthday paradoxon).

9. Databases: Entity-Relationship Model

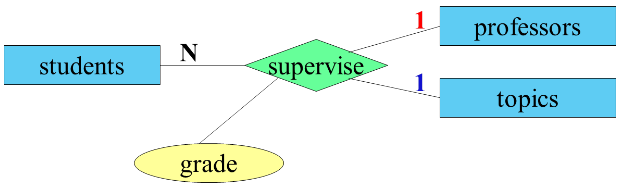
The Entity-Relationship Model provides a means to conceptually model a part of the world for a database in a graphical way.

It consists of *entities* and *relationships* which both can be characterized with *attributes*. Entities can provide a *key attribute* that models a unique identifier of an identity. Relationships can play *roles* with respect to entities.



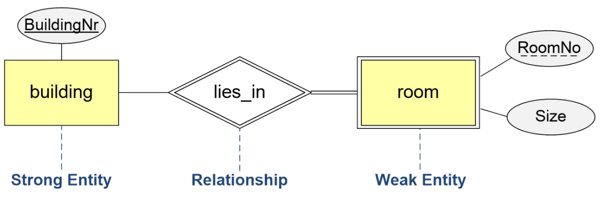
An additional tool in modeling relationships is the specification of how many items may stand in relation to how many other items. Such *functionalities* are provided in the form "1:1", "1:N", "N:M" for binary relations, and "1:N:M:..." for n-ary relations. A functionality containing a "1" somewhere provides the information that the relationship can be written as a partial function with codomain of the entity type at the "1".

If, for eample, entities e_1, e_2 and e_3 are related as "1:N:1", then their relation is a subset $R \subset E$ of the cartesian product of their domains $E = E_1 \times E_2 \times E_3$ and can be understood as both partial function $f_R : E_2 \times E_3 \nrightarrow E_1$ and as partial function $g_R : E_1 \times E_2 \nrightarrow E_3$. This often helps with the interpretation of relationships.



supervise: professors $\times$ students $\nrightarrow$ topics
supervise: topics $\times$ students $\nrightarrow$ professors

We introduced *weak relationships* where the *weak entity*, marked by double lines, cannot exist without the *strong entity* and can only be identified uniquely with its associated strong entity. Functionality is always $1 : N$ or $1 : 1$.



10. Databases: Relational Model

Using the relational model a database is described as a collection of *relations* (tables) $R \subset D_1 \times \cdots \times D_n$ over domains D_i of *attributes*. A *tuple* $t \in R$ constitutes an element of a relation, i.e. a row in a table. *Schemas* of such tables are described in the form

$$\{[A_1, \ldots, A_n]\}$$

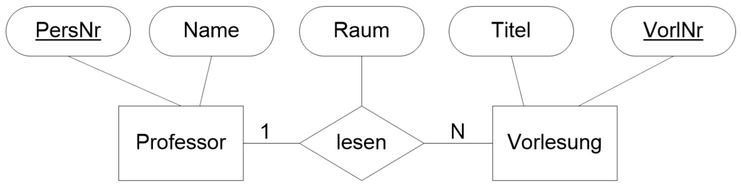
where attributes A_i are usually described in a name:domain form. A *key* is a minimal set of attributes that uniquely identifies a tuple in a relation.

When translating from an ER-Model to a relational model, entity attributes translate to relation attributes. Key attribute translate to *primary keys* of relations. When translating from a relationship, the attributes of all entities together with the attribute of the relationship form the attributes of the relation

$$\{[A_{11}, \ldots, A_{1n_1}, A_{21}, \ldots, A_{2n_2}, \ldots, A_{m1}, \ldots, A_{mn_m}, A_1^R, \ldots, A_{n_r}^R]\}$$

The key of the relation is then formed by all keys of all attributes plus the keys of the relationship. A renaming of attributes may be necessary.

After entities and relationships have been translated into relations, (only) relations with the same key can be *merged*. This implies that relations from $N : M$ relationships cannot be merged.



Professor {[<u>PersNr</u>, Name]}, Vorlesung {[Titel, <u>VorlNr</u>]}, lesen {[PersNr, <u>VorlNr</u>, Raum]}
$\Rightarrow$ Professor {[<u>PersNr</u>, Name]}, Vorlesung {[Titel, <u>VorlNr</u>], gelesenVon, Raum},

We discussed the following operations from relational algebra:

1. *Selection* operation $\sigma_p(R)$ selects tuples (rows) from a relation (table) R that fulfil the selection predicate p.

2. *Projection* operation $\pi_a(R)$ projects to the named attributes (columns) a of a relation (table) R

3. *Cartesian product* operation $R_1 \times R_2$ yields all possible pairs $r_1 r_2$ of tuples of R_1 and R_2

4. *Renaming* operation $\rho_S(R)$ assigns a new name S to a relation R, renaming $\rho_{A_1 \to B_1}(R)$ renames an attribute (column) of a relation (table).

5. The *Join* operation $R \bowtie S$ selects tuples from $R \times S$ that have equal values on all attributes with the same names and merges attributes with same names.

6. The *Theta-Join* operation $R \bowtie_\theta S$ coincides with $\sigma_\theta(R \times S)$.

11. Databases: SQL

SQL (Structured Query Language) is a language used in order to define, manipulate and formulate queries on data bases. In terms of querying it provides a mapping of relational algebra to a formalized natural language.

SQL defines various *data types* for typical database entries such as characters, numbers, dates, raw data. As Data Definition Language (DDL), SQL provides statements such as `create table`, `drop table`, `alter table`. As Data Manipulation Language(DML), SQL provides statements such as `insert into`, `delete` and `update`.

A typical simple *query* in SQL looks like

```
select s.Name, v.Titel
from Studenten s, hoeren h, Vorlesungen v
where s. Legi= h.Legi and h.VorlNr = v.VorlNr
```

A select statement in SQL corresponds to projection, the from part specifies (cartesian product of) participating tables and the where clause provides a selection predicate.

Aggregate functions can be used in order to compute aggregate values over columns of a table, returning a single value per column. If aggregation is used together with *grouping*, it returns a tuple for each group.

```
select v.gelesenVon, p.Name, sum(v.KP)
from Vorlesungen v, Professoren p
where v.gelesenVon = p.PersNr and p.Rang = 'FP'
group by v.gelesenVon, p.Name
    having avg(v.KP) >= 3
```

For nested statements and grouping, it is important to understand possible execution orders. The previous query is executed as

1. from Vorlesungen, Professoren where gelesenVon = PersNr and Rang = FP

2. group by gelesenVon, Name

3. having avg (KP) >= 3

4. select gelesenVon, Name, sum (KP)

It is possible to use queries in a *nested* way, i.e. the result of a query can serve as table within a query. This can provide correlations between nested and enclosing queries, potentially leading to computationally expensive operations. Constructs such as `exists` or `in` can be used in order to reason about query results within an other query.

```
select s.*
from Studenten s
where exists
  (select p.*
  from Professoren
  where p.GebDatum > s.GebDatum);
```

12. Databases: Programming Interface

JDBC provides a means of accessing data bases from within Java.

Opening a connection requires that a JDBC *driver* is installed for the respective data base type (such as MySQL). Access to the database is then established with a *connection* method under the provision of database url, name and password.

```
try{ Class.forName("com.mysql.jdbc.Driver"); }
catch (ClassNotFoundException e) {... }

Connection conn = null;
try{ conn = DriverManager.getConnection(url, username, password); }
catch (SQLException ex) { ... }
```

SQL statements are executed as strings, i.e. the compiler does not check syntactic or even semantic correctness of SQL statements. Such checking is taking place only during runtime. Therefore data base access code is usually very much "polluted" with exception handling.

Another important difference in comparison to using SQL "natively" is that a general purpose programming language such as Java provides only a sequential way to access data in a table. Therefore the obvious concept of a *cursor* (`ResultSet`) is provided as interface to the set of data.

```
...
try{
  stmt = conn.createStatement();
  ResultSet result = stmt.executeQuery("select name from professoren");

  if (result != null)
        while (result.next())
                        System.out.println(result.getString(1));
}
catch (SQLException ex) {
  System.out.println("SQLException: " + ex.getMessage());
  System.out.println("SQLState: " + ex.getSQLState());
  System.out.println("VendorError: " + ex.getErrorCode());
}
finally {
  ... result.close();   ... stmt.close();
}
```

Prepared statements provide a way to parameterize queries: they can be used when a query is executed more than once with different arguments.

```
PreparedStatement s =
  conn.prepareStatement("SELECT name FROM professoren WHERE rang = ?");
ResultSet r;

s.setString(1, "AP");
r = s.executeQuery();
while (r.next()) ...
```