Sie können den Unterschied zwischen einer Klasse und einem Objekt erklären.

Sie kennen die Grundkonzepte von Abstraktion und Modularisierung.

Sie können Klassen- und Objektdiagramme interpretieren.

Sie können Beziehungen zwischen Klassen erkennen und gemäss Lehrbuch (BlueJ) darstellen.

Klassen sind Baupläne für Objekte (Bauplan eines Wagens), in einer Klasse sind die Eigenschaften und das Verhalten definiert die die Objekte haben können. Eine Klasse kann mehrere Objekte (Instanzen) erzeugen.

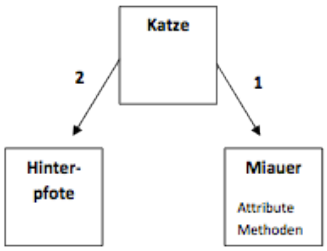
Objekte sind erzeugte Gegenstände (roter Wagen, gelber Wagen), sie haben Attribute (Farbe, Marke...) Wir programieren Klassen (Baupläne, Rezepte), während der Laufzeit werden beliebig viele Objekte erstellt

Abstraktion bezeichnet ein Vorgang, bei dem ein Modell in ein vereinfachtes Modell gefasst wird. Man konzentriert sich auf das Wesentliche und vernachlässigt unwichtige Details. Abstraktion ist von der Perspektive des Betrachters abhängig. (Abspielen einer CD -> stereo-mono, Lautstärke)

Modularisierung bezeichnet die Zerlegung einer Gesamtaufgabe in Teilaufgaben und die Definition der erforderlichen Schnittstellen. Sie dient zur Reduktion der Systemkomplexität. (Stereoanlage -> Tuner, CD-Deck, Boxen, Tape)

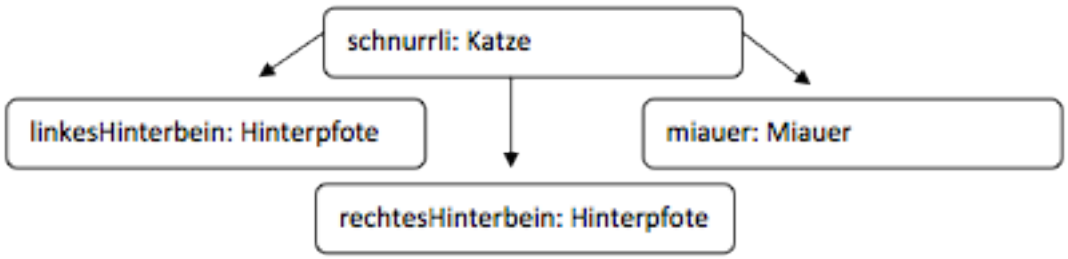
Klassendiagramme:

- zeigen Klassen und deren Beziehung untereinander
- zeigen die statische Sicht eines Programms
- halten Informationen über den Source-Code fest



Objektdiagramme:

- zeigen Objekte zu einem bestimmten Zeitpunkt im Programmablauf
- zeigen die dynamische Sicht eines Programms zur Laufzeit
- stellen Beziehungen zwischen den Objekten dar (welches Objekt referenziert / benutzt andere Objekte)



Sie kennen das Konzept des Information Hiding und wissen, wie es durch Zugriffsmodifizierer unterstützt wird.

Sie kennen die Begriffe Kohäsion und Kopplung.

Sie wissen, wie sich Kopplung im Source Code manifestiert

- Als Datenkapselung (information hiding) bezeichnet man das Verbergen von Daten oder Informationen vor dem Zugriff von aussen. Der direkte Zugriff auf die interne Datenstruktur wird unterbunden und erfolgt statt dessen über definierte Schnittstellen.
- Idee: Eine Programmiererin / ein Programmierer muss nur wissen, welche Methoden eine Klasse anbietet (die «Interface»-Dokumentation). Sie / Er braucht die Implementationsdetails nicht zu kennen (vgl. Modularisierung und Abstraktion).
  → no need to know
- Ein weiterer wichtiger Vorteil von Information Hiding ist eine lose Kopplung zwischen der selber programmierten und der verwendeten Klasse.
  → not being allowed to know
- begünstigt starke Kohäsion und lose Kopplung
- Ein wichtiges Mittel für das Information Hiding ist der Zugriffsmodifizierer private. Attribute sollten immer private sein.
- Methoden, die Verhalten nach aussen zur Verfügung stellen, sollten public sein.
- Es sollte möglichst wenig public Methoden geben.
- Methoden werden als private definiert, wenn sie nur von anderen Methoden derselben Klasse aufgerufen werden. In der Regel wurden sie erzeugt, um
  - Verhalten, das verschiedene Methoden verwenden, nur einmal zu implementieren (Vermeidung von Codeduplikation).
  - den Code besser lesbar zu machen (statt einer «grossen» Methode eine Methode, die andere verwendet).

Als Kohäsion wird der innere Zusammenhalt einer Klasse bezeichnet.
Als Kopplung wird der Zusammenhang zwischen den Klassen bezeichnet.
Das Ziel von einem guten Klassendesign ist eine **lose** Kopplung und eine **starke** Kohäsion.

Loose Kopplung
Das Klassendiagramm zeigt nur wenige Beziehungen zwischen den Klassen, d.h. die Klassen sind möglichst unabhängig voneinander.

Starke Kohäsion
1 Klasse repräsentiert 1 logische Einheit.
1 Methode repräsentiert 1 zusammenhängende Aufgabe.
Jede Klasse handhabt ihre Daten möglichst selber.
Es liegt möglichst wenig Code-Duplikation vor.
Potenzielle Änderungen haben nur lokale Auswirkungen.
Nur eine Klasse bzw. möglichst wenige Klassen kapseln Informationen betreffend des User- Interfaces.

Sie kennen je 3 Folgen von gutem und von schlechtem Klassendesign.

Sie verstehen das Konzept der Polymorphie von Klassentyp-Variablen.

Sie können das Konzept der Vererbung anwenden und kennen Vor- und Nachteile.

Folgen von schlechtem Klassen-Design

- langsame Entwicklung
- schwierige Entwicklung
- fehlerhafte Entwicklung
- abgebrochene Entwicklung
- nachträgliche Änderungen schwierig umsetzbar

Folgen von gutem Klassen-Design

- Lesbarkeit
- Testbarkeit
- Wiederverwendung

Variablen eines Klassentyps in Java sind polymorph

- Sie können sowohl Objekte des deklarierten Types aufnehmen wie auch alle Objekte aller abgeleiteten Subtypen.

Vererbung (vereinfacht)

- Eine abgeleitete Klasse besitzt («erbt») Methoden und Attribute der Basisklasse
- Sie kann auf diese zugreifen (falls nicht durch Zugriffsmodifizierer eingeschränkt)
- Schlüsselwort **extens** definiert die Vererbungs-Beziehung
- Schlüsselwort **super** verwendet im Konstruktor der Unterklasse

Vorgängig zum Kreieren des Objekts der Unterklasse muss zuerst das Objekt der Basisklasse initialisiert (erzeugt) werden.
Erzeugung erfolgt automatisch, falls Basisklasse einen parameterlosen Konstruktor hat (sonst Syntax-Fehler)
andernfalls muss die erste Anweisung im Konstruktor der Unterklasse super sein

```
public class Unterklasse extends Basisklasse {

    public Unterklasse()
    {
        super();
    }
}
```

Vorteile

- hilft, Code-Duplikation zu vermeiden
- unterstützt Code-Wiederverwendung
- vereinfacht den Unterhalt
- unterstützt einfache Erweiterungen bestehender Klassen

Nachteile

- erhöht die Kopplung (neue Abhängigkeiten zwischen den Unterklassen und der Basisklasse)

Sie können das Konzept des Subtyping, insbesondere das entsprechende Handling von Referenzvariablen, nutzen.

Sie kennen den prinzipiellen Aufbau einer Klasse.

Sie wissen, wozu Instanzvariablen, Methoden und Konstruktoren dienen.

Subtypen

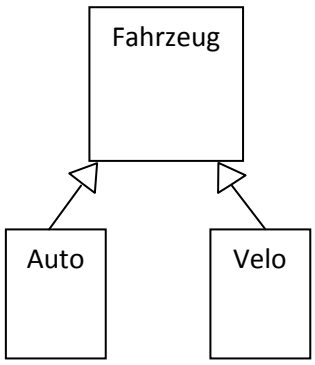
- Analog zur Klassenhierarchie bilden Klassentypen ebenfalls eine Hierarchie
- Der Klassentyp einer Unterklasse ist ein Untertyp (Subtyp) des Basis-Klassentyps.

Variablen und Subtypen

- Variablen referenzieren ein Objekt des deklarierten Typs oder auch Objekte eines Subtyps des deklarierten Typs

Substitution

- Subtypen Objekte können verwendet werden, auch wenn ein Basisklassen Objekttyp erwartet wird




```
Fahrzeug f1 = new Fahrzeug();
Fahrzeug f2 = new Auto();
Fahrzeug f3 = new Velo();

//falsch
Auto a2 = new Fahrzeug();
```

```
Fahrzeug fahrzeug1;
Fahrzeug fahrzeug2;
Auto auto;
Velo velo;

public void ObjektErstellen()
{
//implizit
auto = new Auto();
fahrzeug1 = auto;

//explizit
auto = (Auto)fahrzeug1;

//Compiler-Fehler (Subklasse - Subklasse)
velo = (Velo)auto;

//Laufzeit-Fehler
fahrzeug2 = auto;
velo = (Velo)fahrzeug2;
```

```
public class Manu {

	// 1. Instanzvariablen -> Eigenschaften eines Objekts
	private int var1;

	// 2. Konstruktor	-> Initialisierung eines Objekts
	public Manu ()
	{
	}

	// 3. Methoden		-> Verhalten eines Objekts
	private void setVar1()
	{
	}
}
```

Methoden dienen zur Abstraktion und Wiederverwendung
Sie realisieren das Verhalten von Objekten
Sie ermöglichen den Zugriff und die Kommunikation/Interaktion unter Objekten

Sie kennen mindestens je 3 Eigenheiten von Instanzvariablen, Klassenvariablen, lokalen Variablen und von formalen Parametern.

Sie können die Rollen von Methodenkopf und Methodenrumpf erklären.

Instanzvariablen:
- halten die Eigenschaften eines Objekts fest (Durchmesser, Farbe)
- existieren zeitlebens eines Objekts
- sind innerhalb der ganzen Klasse sichtbar/ansprechbar

Klassenvariablen:
- werden mit `static` deklariert
- sind für die Klasse nur einmal vorhanden (nicht wie Instanzvariablen für jedes Objekt)
- existieren zeitlebens einer Klasse

lokale Variablen:
- sind temporärer Speicher
- werden innerhalb einer Methode bzw. Blocks deklariert
- existieren zeitlebens eines Blocks

Formale Parameter:
- sind im Header einer Methode / eines Konstruktors definiert
- beim Aufruf einer Methode / eines Konstruktors müssen sie übergeben werden
- existieren zeitlebens der Methode / des Konstuktors
- sind nur innerhalb der Methode / des Konstruktors sichtbar
- sind spezielle lokale Variablen

Methodenkopf: public int getColor() WAS
- Zugriffsmodifizierer typisch `private` oder `public`
- `void` entspricht keiner Wertrückgabe, sonst einer der 8 elementaren Datentypen (`return`)
- Signatur besteht aus Methodenname und Parameter `setColor(int 233)`

Methodenrumpf: alles innerhalb `{..}` WIE
- beinhalten Anweisungen und Deklarationen
- Rückgabewert wird mit `return` zurückgegeben

Sie können einen Konstruktor-Kopf ohne/mit Parameter(n) formulieren.

Sie können einen Methoden-Kopf ohne/mit Parameter(n) bzw. ohne/mit Rückgabewert formulieren.

Sie können überladene Konstruktoren und Methoden erstellen.

Sie können Accessor Methoden formulieren.

Sie können Mutator Methoden formulieren.

Sie können den Unterschied zwischen einem formalen und einem aktuellen Parameter erklären.

```
public class Test{
     private int position;
     public Test(){
          position = 0;
          altitude = 0;
     }
     public Test(int i){
          position = 1;
          altitude = 1;
     }

     public void setLocation(int pos,int alti){
          position = pos;
          altitude = alti;
     }
     public void start(){

     }
     public int rechnen(int a,int b){
          return(a*b);
     )
     public int rechnen(int a,int b, int c){
          return(a*b*c);
     }
```

Accessor Methode
Accessor-/getter- oder Zugriffs-methoden ist eine Methode die den Wert eines Attributs zurückgeben, der Vorteil ist die Entkoppelung der internen Darstellung.

```
public boolean istKanalOffen()
{
    return status;
}
```

Mutator Methode
Mutator-Methoden dienen zur Zustandsänderung eines Objektes, sie muss nichts zurückgeben aber kann Bsp.: `public int zurueckgebenUndAendern()`
Setter-Methoden sind spezielle Mutator-Methoden, durch sie werden Werte von Attributen geändert Bsp.: `public void setColor()`

```
public void schliessen()
{
    status = false;
}
```

Unterschied zwischen formalem und aktuellen Parameter

```
public void starten()
{
     rechnen(10, 12);
}
```

aktuelle Parameter

```
public void rechnen(int zahl1, int zahl2)
{
    resultat = zahl1 * zahl2;
}
```

formale Parameter

Sie können Instanzvariablen und lokale Variablen deklarieren.

Sie wissen, wie man mit Objekten arbeitet. JA!

Sie wissen, wieso und wie man Klassen importiert.

Sie können Klassenvariablen definieren.

Sie können Klassenmethoden definieren.

Sie können Konstanten und Klassenkonstanten definieren.

Sie können eine main()-Methode implementieren.

- Instanzvariablen

```
public class Test {

    private int position;
}
```

- lokale Variablen

```
public int getVolume()
{
    int r3;
    r3 = 1 + 2;
    return r3;
}
```

Klassen-Bibliothek hat viele Klassen
--> benötigt Struktur
Einteilung von zusammengehörigen Klassen in Pakete (package) und Unterpakete

```
import java.util.ArrayList;
import java.util.*;
```

Klassenmethoden:

- Klassen-Methoden werden in der Klasse und nicht in einem Objekt gespeichert, daher kann man sie aufrufen ohne zuvor ein Objekt erstellt zu haben
- In statischen Methoden besteht kein Zugriff auf nicht statische Attribute
- In statischen Methoden dürfen keine nicht-statischen Methoden aufgerufen werden.

```
public static double roundScale(double d)
{
    return Math.rint(d * 100) / 100;
}
```

Klassenvariablen:
Klassen-Variablen werden in der Klasse und nicht in einem Objekt gespeichert, daher kann man sie aufrufen ohne zuvor ein Objekt erstellt zu haben

```
public static PrintStream out;
```

Konstanten und Klassenkonstanten:

```
public static final double PI = 3.141592653589793;
private final int i = 7;
```

main()-Method

- Beim Start der Klasse wird immer zuerst die Main-Methode aufgerufen
- In dieser Methode können dann Objekte erzeugt und weitere Methoden aufgerufen werden.

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

}
```

Sie verwenden die Interfacebeschreibung einer Klasse (vgl. API-Dokumentation), um sie in Ihren Programmen zu nutzen. DONE!

Sie können einfache Klassenhierarchien in Java implementieren.

Sie verstehen die Bedeutung der Klasse Object und kennen überblicksmässig deren Basisfunktionalität.

Sie kennen die vier Zugriffsmodifizierer und wissen, wie man sie anwendet.

Sie können die acht primitiven Datentypen benennen und einordnen.

Sie können vergleichende Bedingungen formulieren und logische Operatoren anwenden.

Sie können arithmetische und logische Ausdrücke auswerten. DONE & üben!

Klassenhierarchie

```
public class Fahrzeug {

}
public class Velo extends Fahrzeug {

}
public class Auto extends Fahrzeug {

}
```

- Implizit ist die Klasse Objekt die Basisklasse für alle Klassen.
- Somit ist jede Klasse eine Unterklasse der Klasse Objekt.

Zugriffsmodifizierer regeln die Sichtbarkeit von Klassen, Methoden und Variablen

	Eigener Klasse	Klasse im gleichen Package	Unterklasse aus anderem Package	Nicht-Unterklasse aus anderem Package
private	Ja	Nein	Nein	Nein
-	Ja	Ja	Nein	Nein
protected	Ja	Ja	Ja*	Nein
public	Ja	Ja	Ja*	Ja*

Schlüsselwort/ Typ	**Länge in Byte**	**Belegung (Wertebereich)**
boolean	1	true oder false
char	2	16-Bit-Unicode-Zeichen (0x0000 ... 0xFFFF)
byte	1	–2^7 bis 2^7 – 1 (–128 ... 127)
short	2	–2^15 bis 2^15 – 1 (–32.768 ... 32.767)
int	4	–2^31 bis 2^31 – 1 (–2.147.483.648 ... 2.147.483.647)
long	8	–2^63 bis 2^63 – 1 (–9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807)
float	4	1,40239846E–45f ... 3,40282347E+38f
double	8	4,94065645841246544E–324 ... 1,79769131486231570E+308

Operator	**Typ**	**Beschreibung**
++, --	arithmetisch	Inkrement und Dekrement
+, -	arithmetisch	unäres Plus und Minus
!	boolean	logisches Komplement
*, /, %	arithmetisch	Multiplikation, Division, Rest
+, -	arithmetisch	Addition und Subtraktion
+	String	String-Konkatenation
<, <=, >, >=	arithmetisch	numerische Vergleiche
==, !=	primitiv	Gleich-/Ungleichheit von Werten
==, !=	Objekt	Gleich-/Ungleichheit von Referenzen
&	boolean	logisches Und
^	boolean	logisches Xor
|	boolean	logisches Oder
&&	boolean	logisches konditionales Und, Kurzschluss
||	boolean	logisches konditionales Oder, Kurzschluss
=	jeder	Zuweisung

Sie können das implizite und das explizite Type-Casting anwenden.

Sie können Arrays deklarieren, initialisieren sowie auf Arrays zugreifen.

Sie können einfache Methoden mit Array-Parametern implementieren und diese aufrufen.

Implizites Typecasting wird von Java selbst durchgeführt:
Bsp.: `kommazahl = ganzeZahl;`
Explizites Typecasting wird vom Programmierer durchgeführt:
Bsp.: `kommazahl = (float) ganzeZahl;`
Auf einen „grösseren" Datentyp kann das implizite Typecasting problemlos angewendet werden.

von	**nach (implizit)**
byte	char, short, int, long, float, double
char, short	int, long, float, double
int	long, float, double
long	float, double

float	double
«alle»	nach String

Arrays:

- Kann Werte eines bestimmten elementaren Datentyps oder Objekts (Referenzen) eines bestimmten Klassentyps aufnehmen.
- Auf die Werte bzw. Objekte kann via Index (ganze Zahl) direkt zugegriffen werden: 0 to length- 1
- Ein Array selbst wird wie ein Objekt gehandhabt.
- Mit dem Erzeugen eines Array wird seine Länge (length) unveränderlich festgelegt (statische Datenstruktur)
- Bei der Ausführung überprüft die Java Virtual Machine vorgängig jeden Array-Zugriff (unzulässig --> Exception)

```
public void array()
{
    int[] a;
    a = new int[5];

    //oder

    int[] b = new int[5];

    a[0] = 1;
    a[1] = 2;

    //Schleife
    for(int i = 0; i < a.length; i++) {
        a[i] = i * 2;
    }

    //Deklaration inkl. Initialisierung
    int[]c = {0, 2, 4, 6, 8};
}
```

Methoden mit Array Parametern

```
public void array()
{
    int[] table = new int[10];

    for(int i = 0; i < table.length; i++) {
        table[i] = i;
    }
    int total = sum(table);
}

private int sum(int[] values)
{
    int s = 0;
    for(int i = 0; i < values.length; i++){
        s = s + values[i];
    }
    return s;
}
```

Sie können die Selektion mit if anwenden.

Sie können eine einfache Selektion mit switch formulieren.

Sie kennen die Eigenheiten der verschiedenen Schleifen-Anweisungen (while, do while, for, foreach).

If- Selection

- Operatoren:

<	kleiner	<=	kleiner oder gleich
==	gleich	!=	ungleich
>	grösser	>=	grösser oder gleich

```
if(expression) {
    //statements
}
else {  //optional
    //statements
}
```

Switch

```
switch(day) {
    case 1: dayString = "Monday";
            break;
    case 2: dayString = "Tuesday";
            break;
    default: dayString = "invalid day";
            break;
}
```

Wenn **break** fehlt werden alle nachfolgenden **case** auch geprüft.

while

Bei der **`while`** Schleife ist die Anzahl Schleifendurchgänge unbekannt. Es kann auch sein, dass die Schleife nie durchlaufen wird. Die Schleife wird solange ausgeführt bis die Bedingung der Expression **false** lautet.

do – while

Die **`do – while`** Schleife verhält sich gleich wie die **`while`** Schleife bis auf die Eigenschaft, dass die
Schlaufe mindestens einmal durchlaufen wird.

for

Bei der **`for`** Schleife sind die Anzahl Schleifendurchläufe bekannt oder im Voraus (eventuell erst zur Laufzeit) berechenbar. Die **`for`** Schleife erlaubt es, dass unter Umständen kein einziger Schleifen- Durchlauf ausgeführt wird. In der Initialisierung und Finalisierung der **`for`** Schleife dürfen auch mehrere Anweisungen enthalten sein.

for – each

Die **`for – each`** Schleife ermöglicht das einfache Zugreifen auf alle Elemente einer Collection oder Arrays. Allerdings kann auf Werte eines primitiven Datentyps nur lesend zugegriffen werden.

Sie können für eine gegebene Aufgabenstellung die geeignete Schleifen-Anweisung bestimmen. JA!

Sie können einfache Methoden mit Schleifen-Anweisungen implementieren.

Sie können Klassen aus der Klassenbibliothek importieren. (Bereits beantwortet)

Sie können Variablen des Typs String deklarieren.

Sie können die String-Konkatenation mit + ausführen.

Sie können Strings vergleichen.

Sie verwenden die Methode `System.out.println()`. JA!

while

```
while(expression) {
     //statements
}
```

do – while

```
int nr = 0;
do {
     nr = (int)(Math.random() * 6) + 1;
}
while(!(nr == 6));
```

for

```
int result = 1;
for(int i = 0; i < n; i++) {
return result * i;
}
```

for – each

```
char [] a = {'a', 'b', 'c', 'd', 'e');
for(char ch : a) {
System.out.println(ch);
}
```

```
String text = „test";

String text 2 = text + „2";

String neuesWort = „Weihnachts" + text + „mann";
```

Strings dürfen nicht mit == verglichen werden, da Strings nur Speicherreferenzen sind, der Vergleich muss mut `.equals()` gemacht werden.

```
String name = „Hans";
if(name.equals(„Hans")){ //Test auf Inhalt
}
if(name == „Hans"){ //Test auf Referenz
}
```

Sie kennen überblicksmässig das Konzept der Wrapper-Klassen (z.B. Integer, Long, Float).

Sie können Wrapper Klassen anwenden und verstehen den Autoboxing / -unboxing Mechanismus.

Sie verstehen "Generics" und können sie anwenden.

Sie kennen die Container-Klassen überblicksmässig.

Sie können die wichtigsten Methoden der Klassen List/ArrayList, Set/HashSet und Map/HashMap anwenden.

Zu jedem primitiven Datentyp in Java gibt es eine korrespondierende **Wrapper-Klasse**. Diese kapselt die primitive Variable in einer objektorientierten Hülle und stellt eine Reihe von Methoden zum Zugriff auf die Variable zur Verfügung.

Bsp. ArrayList mit float Objekten:

```
ArrayList floats = new ArrayList(); // ArrayList ohne Generics

float f = 4.23;                          // float Variable f
Float floatObject;                  // Klasse Float

floatObject = new Float(f);         // in Float Objekt einpacken
floats.add(floatObject);            // einfügen

floatObject = (Float)floats.get(0);// auslesen
f = floatObject.floatValue();       // konvertieren Object in
Zahl
```

- Durch die Generics in Java entfällt das manuelle Einpacken / Auspacken (**boxing** / unboxing) der primitiven Datentypen.
- Der Compiler hat genug Informationen und kann das wrapping / unwrapping automatisch übernehmen und die Typen überprüfen.Bsp.:

```
ArrayList<Float> floats = new ArrayList<Float>();
float f = 3.2f;
floats.add(f); //einfügen, Compiler übernimmt boxing
f = floats.get(0); //auslesen, Compiler übernimmt unboxing
```

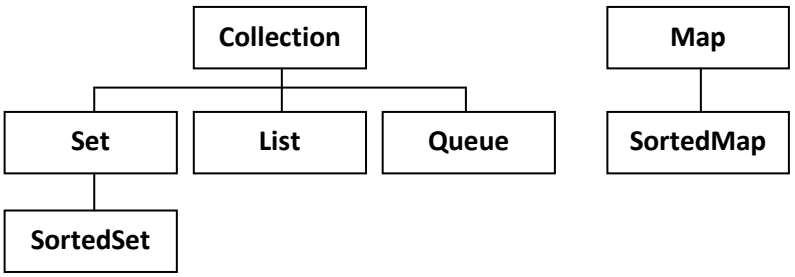
Konzept der Generics

Durch die Generics in Java entfällt das manuelle Einpacken / Auspacken (boxing / unboxing) der primitiven Datentypen. Mit Generics werden ein oder mehrere Datentypen definiert.

z.B. **`ArrayList`** für bestimmte Objekte

```
ArrayList <String> strings = new ArrayList<String>();
```

Container-Klassen



Mehr unter:
HSLU/Semester1/Programmieren1/Zusammenfassungen/Java_Collections_Zusmf.pdf

Sie kennen das Konzept der Iteratoren und können dieses anwenden.

Sie kennen verschiedene Möglichkeiten, um alle Elemente einer Collection (z.B. ArrayList) traversieren bzw. bearbeiten zu können (for, foreach, Iterator).

- erlauben das Traversieren einer Collection ohne Bezug auf den Index zu nehmen (falls überhaupt vorhanden)
- jedes Element wird während einer Iteration genau einmal zurückgegeben
- bemerkt in der Regel (aber Vorsicht) eine Veränderung der Collection
- während eine Iteration läuft
- zwei Arten
  - Iterator
    - `boolean hasNext()`
    - `E next()`
    - `void remove()`

```
ArrayList<Float> floats = new ArrayList<Float>();
Iterator<Float> iterator;
float fl;

iterator = floats.iterator();
while(iterator.hasNext()) {
    //Statements
    fl = iterator.next();
}
```

  - ListIterator
    - wie Iterator, aber zusätzlich
    - boolean hasPrevious()
    - E previous()

```
int[] a = new int [5];
int wi = 0; int di = 0;

while(wi < a.length){
      a[wi] = wi;
      wi++;
}
for(int fi = 0; fi < a.length; fi++){
      System.out.println(„Position“ + a[fi]);
}
do{
      System.out.println(„Position „ + a[di]);
      di++;
} while(di < a.length);

ArrayList list = new ArrayList();
for(int = 0; i <20; i++){              //füllen mit for-Schleife
      list.add(„Objekt „ + i);
}
for(int i=0; i< list.size(); i++){     //auslesen mit for-Schleife
      System.out.println(list.get(i));
}
Iterator it = list.iterator();         //auslesen mit Iterator
while(it.hasNext()){
      System.out.println(it.next());
}
for(String k:list){ //auslesen mit for-each-Schleife(string=Objekt)
      System.out.println(k); }
```

Sie kennen die Merkmale eines Algorithmus.

Sie können den Begriff der Komplexität eines Algorithmus definieren.

Verfahren, mit dem eine Problemkategorie gelöst werden kann:

- schrittweises Verfahren
- nächster Schritt eindeutig
- ausführbare Schritte
- endet nach endlich vielen Schritten

Computer ist prädestiniert, Algorithmen auszuführen
zentrale Themen der Informatik und Mathematik

- Algorithmentheorie
- Komplexitätstheorie
- Berechenbarkeitstheorie

eigenes Gebiet in der Informatik
Beispiele von Algorithmen

- Suchverfahren
- Sortieralgorithmus (Bubblesort, Quicksort)
- Berechnung von Schaltjahren
- zyklische Redundanzprüfung (CRC)
- kryptographisches Verfahren (z.B. DES / 3DES)
- Audio- / Video-Kompression (MPEG)
- «Berechnung» von Zufallszahlen

Komplexität (auch Aufwand oder Kosten) eines Algorithmus:

- maximaler Ressourcenbedarf in Abhängigkeit der Eingabedaten (häufig Grösse der Datenmenge oder Werte)

Ressourcen sind:

- Anzahl benötigter Rechenschritte (Zeitkomplexität)
- Speicherbedarf (Speicherkomplexität)

Wie wächst der Ressourcenbedarf, wenn mehr Daten zu verarbeiten sind?
--> verdoppelt oder quadriert

- kein Interesse am präzisen Aufwand eines konkreten Programms auf einem bestimmten Computer

Sie verstehen den Begriff "Ordnung eines Algorithmus" und können die Ordnung einer gegebenen Funktion bestimmen.

Sie kennen das Prinzip der Rekursion.

Sie können die Rekursionsbasis und Rekursionsvorschrift bestimmen.

Sie können einfache rekursive Methoden implementieren.

Abhängigkeit aus Analyse des Algorithmus
jedoch kein Interesse am genauen Funktionsverlauf f(n), nur an der Ordnung O

- Ordnung ignoriert konstante Faktoren
  - diese können praktisch relevant sei
- Ordnung berücksichtigt Verhalten bei kleinen n nicht
  - langsamere Algorithmen bei kleinen n oft brauchbar / sinnvoll
- Die exakte mathematische Analyse von vielen Algorithmen ist schwierig oder unmöglich
- Algorithmen mit exponentieller Ordnung dauern trotz schnellen Computern oft zu lange
  - ➔ Suche nach schnelleren Algorithmen

$f(n) = n^3 + 0.1*2^n$	$f(n) = 2 + 37*n^3 + 0.01 * n^4$
$f(n) = O(2^n)$	$f(n) = O(n^4)$
$f(n) = 1000*n + n^3$	$f(n) = 5326 + \ln(n)$
$f(n) = O(n^3)$	$f(n) = O(\ln(n))$
$f(n) = \ln(n) + 1000*n$	$f(n) = n^7 + n!$
$f(n) = O(n)$	$f(n) = O(n!)$

Konstant	$O(1)$
Logarithmisch	$O(\ln(n))$
Linear	$O(n)$
n log n	$O(n * \ln(n))$
Polynomial ①	$O(n^m)$
Exponential	$O(d^n)$
Fakultät	$O (n !)$

- Eine Funktion / Methode heisst rekursiv, wenn sie sich (direkt oder indirekt) selber aufruft
- Eine Rekursion besteht aus einer Rekursionsbasis und einer Rekursionsvorschrift
- Die **Rekursionsbasis** beschreibt, wie ein einfaches Problem der angegebenen Problemklasse gelöst werden kann
- Die **Rekursionsvorschrift** gibt an, wie ein Problem auf ein einfacheres Problem zurückgeführt werden kann. Innerhalb der Rekursionsvorschrift wird die Funktion / Methode selber wieder aufgerufen

```
static void permute(char[] a, int endIndex)
{
    if (endIndex == 0) {                    // Rekursionsbasis
        System.out.println(a);
    }
    else {                                  // Rekursionsvorschrift
    for (int i=0; i <= endIndex-1; i++)  {
        exchange(a, i, endIndex);
        permute(a, endIndex-1);
        exchange(a, i, endIndex);
    }
    permute(a, endIndex-1);
}
```

Beispiele: Fakultät, Pascalische- Dreieck. ÜBEN!!!!

```
public long fak(int n)
{
    if ((n == 1) || (n == 0)){
        return 1;
    }
    else {
        return n * fak(n-1);
    }
}
```

Sie wissen, wie eine rekursive Methode abgearbeitet wird.

Sie können das Konzept "Rekursion" an einem Beispiel erklären, z.B. mit den Türmen von Hanoi.

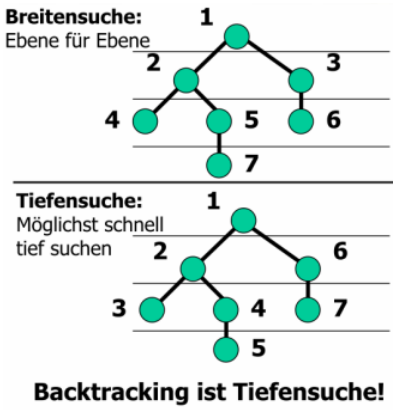
Sie können das Konzept "Backtracking" erklären und anwenden.

- Auf dem Stack im Speicher (Call Stack) befinden sich die Informationen der lokalen Variablen und Parameter der aktuell ausgeführten Methode **`m1()`**.
- Ruft diese Methode **`m1()`** eine andere Methode **`m2()`** auf, so werden zusätzlich die Informationen von **`m2()`** auf dem Call Stack abgelegt (die Informationen von **`m1()`** sind dann nicht sichtbar).
- Jeder neue Methodenaufruf bedeutet, dass mehr Platz auf dem Call Stack benötigt wird.
- Die Rekursion ruft sich so oft auf, bis die Rekursionsbasis erreicht wird und arbeitet sich danach „rückwärts“ wieder ab.

Türmen von Hanoi

```
 * Bewegt n Scheiben von Turm a nach Turm c und benutzt als
 * Zwischenspeicher Turm b.
 */
private static void bewege (char a, char b, char c, int n)
{
if (n == 1)
    System.out.println("Lege die oberste Scheibe von " +
       "Turm " + a + " auf Turm " + c + ".");
else {
     bewege(a, c, b, n-1);
     bewege(a, b, c, 1);
     bewege(b, a, c, n-1);
    }
}
public static void main (String[] args)
{
    bewege('a', 'b', 'c', 4);
}
```

Backtracking ist eine
- systematische Art der Suche in einem vorgegebenem Lösungsraum
- Wenn eine Teillösung in eine Sackgasse führt, dann wird der jeweils letzte Schritt rückgängig gemacht («back-tracking»)



Sie können an einem Beispiel erläutern, was die Eigenschaft "Stabilität" eines Sortieralgorithmus bedeutet.

Sie kennen die Ordnungen der behandelten direkten/einfachen und höheren Sortieralgorithmen.

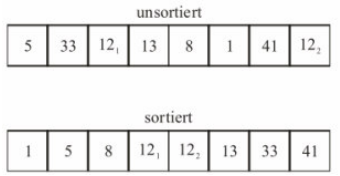
Sie können die Arbeitsweise der behandelten direkten/einfachen und höheren Sortieralgorithmen erklären und an einfachen Beispielen aufzeigen:

Sie kennen die Vor- und Nachteile der behandelten Sortieralgorithmen.

Direktes Einfügen (Insertion Sort)

Falls mehrere Objekte mit gleichem Schlüssel vorliegen

- Reihenfolge dieser Objekte vor und nach dem Sortieren
  gleich --> stabiler Sortieralgorithmus
  ungleich --> unstabiler Sortieralgorithmus
- stabil:



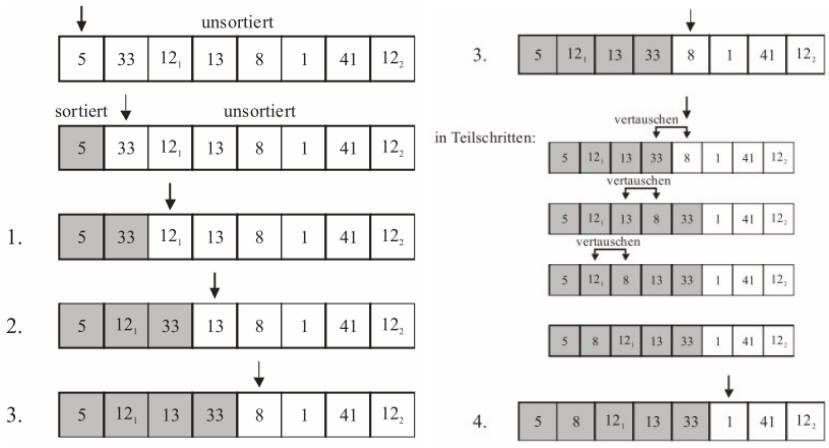
- stabile Algorithmen: direktes Einfügen, Bubblesort, Mergesort
- instabile Algorithmen: direktes Auswählen, Shellsort, Quicksort

Einfache / Direkte Sortieralgorithmen	--> Aufwand O(n_2)
Höhere Sortieralgorithmen	--> Aufwand O(n * log n)

Direktes Einfügen/ Insertion Sort	O (n_2)
Direktes Auswählen/ Selection Sort	O (n_2)
Direktes Austauschen / Bubblesort	O (n_2)
Shellsort	ungefähr O (n_2) tendenziell weniger
Quicksort (höherer Sortier Alg.)	O (n * log(n))
Mergesort (höherer Sortier Alg.)	O (n * log(n))

Direktes Einfügen (Insertion Sort)

- Im ersten Schritt wird das erste Element als sortiert „gesetzt“.
- Danach wird das neu zu sortierende Element (Nachbar) mit dem bereits sortierten Element
- verglichen – Wenn der Schlüssel nicht zutrifft, werden die Elemente getauscht dies geschieht solange bis die Bedingung erfüllt ist.
- Nach n-1 Durchläufen ist das Array sortiert
- Dieses benachbarte «Sesselrücken» ist sicher nicht effizient! Ideal wäre ein Verschieben über grössere Distanzen. Daher für nicht vorsortierte und eher kleinere Mengen, aber stabil.



Sie können die Arbeitsweise der behandelten direkten/einfachen und höheren Sortieralgorithmen erklären und an einfachen Beispielen aufzeigen:

Sie kennen die Vor- und Nachteile der behandelten Sortieralgorithmen.

Direktes Auswählen (Selection Sort)

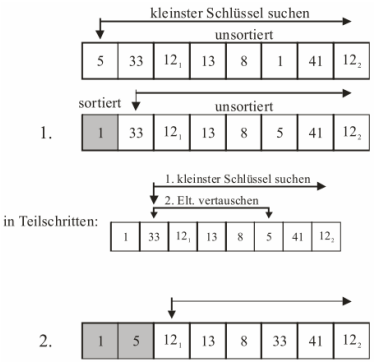
Sie können die Arbeitsweise der behandelten direkten/einfachen und höheren Sortieralgorithmen erklären und an einfachen Beispielen aufzeigen:

Sie kennen die Vor- und Nachteile der behandelten Sortieralgorithmen.

Direktes Austauschen (Bubble Sort)

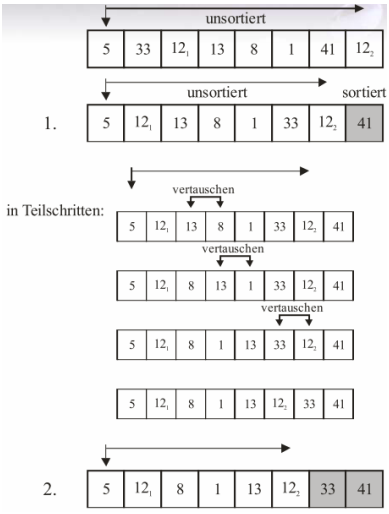
Direktes Auswählen (Selection Sort)

- Das zu sortierende Array wird in einen unsortierten Teil (weiss) und einen sortierten Teil (grau) unterteilt.
- Im unsortierten Teil des Arrays wird das Element mit dem kleinsten Schlüssel ausgewählt bzw. gesucht. Dieses vertauscht man dann mit dem ersten Element des unsortierten Teils.
- Nach n-1 Durchläufen ist das Array sortiert
- Der Aufwand für das direkte Auswählen ist grundsätzlich ebenfalls von der Ordnung (n_2). Das Vertauschen der Elemente über grössere Distanzen wirkt sich aber positiv auf die Effizienz aus, aber instabil, doch wird von links her, also kleinstes Element her geordnet, ideal, wenn das kleinste Element das wichtigste ist



Direktes Austauschen (Bubble Sort)

- Das zu sortierende Array wird in einen unsortierten Teil (weiss) und einen sortierten Teil (grau) unterteilt.
- Zwei benachbarte Elemente werden einfach ausgetauscht, falls deren Schlüssel nicht aufsteigend geordnet ist.
- Das Element mit dem grössten Schlüssel «blubbert» im Verlaufe eines Durchlaufs im unsortierten Teil ganz nach rechts. Der sortierte Teil wird damit um ein Element länger.
- Nach n-1 Durchläufen ist das Array sortiert
- Dieses «Sesselrücken» ist sicher nicht effizient, ist jedoch stabil und bei kleiner Menge geeignet, zudem wenn das grösste (rechts) Element das wichtigste ist.



Sie können die Arbeitsweise der behandelten direkten/einfachen und höheren Sortieralgorithmen erklären und an einfachen Beispielen aufzeigen:

Sie kennen die Vor- und Nachteile der behandelten Sortieralgorithmen.

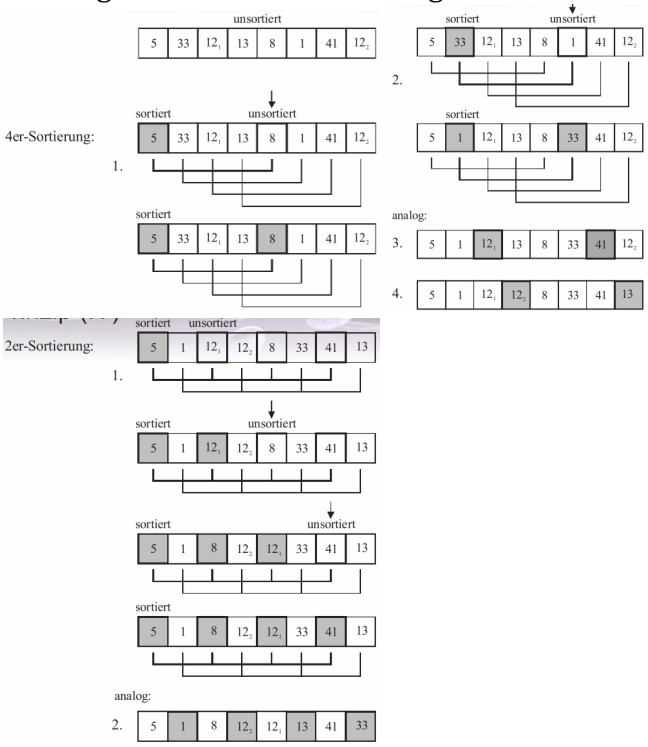
Shellsort

Sie können die Arbeitsweise der behandelten direkten/einfachen und höheren Sortieralgorithmen erklären und an einfachen Beispielen aufzeigen:

Sie kennen die Vor- und Nachteile der behandelten Sortieralgorithmen.

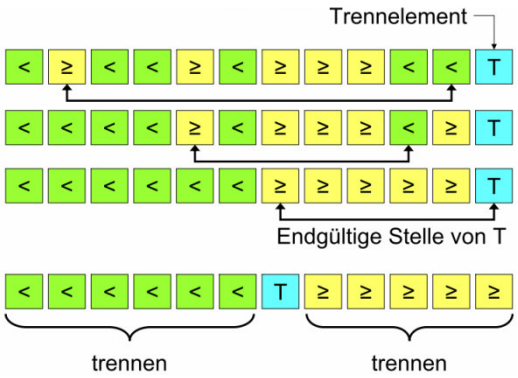
Quicksort

- Bei **Shellsort** erfolgt das Sortieren in mehreren Stufen von «grob» bis «fein».
- Bei der Grobsortierung erfolgt das Sortieren über «grosse Distanzen»; bei der Feinsortierung sind es - falls überhaupt noch notwendig - nur noch «kleine Distanzen».
- Das eigentliche Sortieren erfolgt aber wie beim direkten Einfügen.



Quicksort wird mittels Rekursion beschrieben:

- **Rekursionsvorschrift:** Die zu sortierende Folge wird in zwei Teilfolgen getrennt. Die Teilfolgen werden anschliessend mit dem gleichen Verfahren weiter getrennt.
- **Rekursionsbasis:** Eine Folge von einem Element ist sortiert.
- **Trennverfahren:**
  - Bestimme ein Trennelement
  - Ziele: Das Trennelement steht an seiner endgültigen Position. Alle Elemente rechts vom Trennelement sollen grösser oder gleich diesem sein. Alle Elemente links vom Trennelement sollen kleiner dem Trennelement sein.
- Bestimmen des Trennelementes: Zwei gängige Verfahren:
  - Nimm das letzte Element der Folge.
  - median-of-three: Nimm das Element mit dem mittleren Wert von drei Elementen
- Trennverfahren:
  - Ordne die Folge so um, dass alle Glieder, welche kleiner dem **T** sind, nach links und grössere oder gleiche Elemente nach rechts verschoben werden.
  - Verschiebe (tausche) das **T** an seinen endgültigen Platz.
  - rekursiver Aufruf jeweils getrennt für die linke sowie rechte Teilfolge



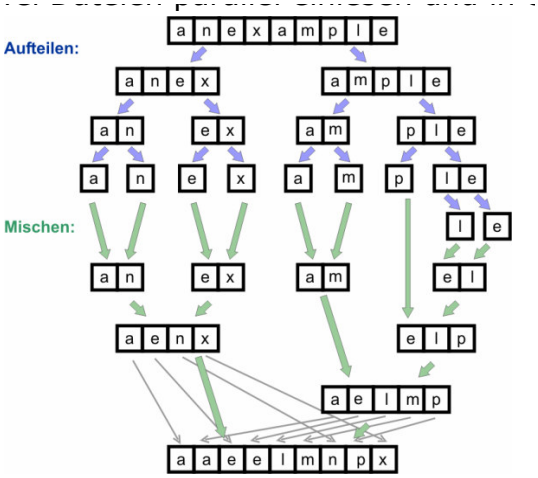
Sie können die Arbeitsweise der behandelten direkten/einfachen und höheren Sortieralgorithmen erklären und an einfachen Beispielen aufzeigen:

Sie kennen die Vor- und Nachteile der behandelten Sortieralgorithmen.

Mergesort

Sie verstehen die im Unterricht betrachteten konkreten Implementierungen von Sortieralgorithmen, z.B. von Quicksort.

Mergesort



- Halbiert den Array immer zur Hälfte. Bis es nur noch zwei Elemente gibt, die kann er dann einfach Austauschen, falls nötig. Danach werden alle Teilelementblöcke wieder zusammengeführt.

- Bei der Aufteilung in kleinere Teilfolgen ist die Grösse jeder Teilfolge ungefähr gleich (+/- 1 Element)

- Binärbaum, wie bei Quicksort im besten Fall

- Zeitkomplexität garantiert immer O(n * log n)
- Das Verfahren benötigt nochmals soviel Speicherplatz wie die Folge (für das Mischen) Speicherkomplexität O(n)
- Mergesort ist gegenüber der ursprünglichen Reihenfolge der Eingabedaten zeit-unempfindlich (speziell bei bereits sortierten Folgen)
- ideal für: sortieren von verketteten Listen, sortieren von Daten auf externen Datenspeichern, vorsortierten und grossen Listen

```
private void exchange(char[] a, int firstIndex, int secondIndex)
{
    char tmp;
    tmp            = a[firstIndex];
    a[firstIndex]  = a[secondIndex];
    a[secondIndex] = tmp;
}
/**
* Quicksort Algorithmus.
* @param a     Array zum Sortieren
* @param left  Linke Grenze, zu Beginn 0
* @param right Rechte Grenze, zu Beginn a.length-1
*/
public void quickSort(char[] a, int left, int right)
{
    int     up         = left;      // linke Grenze
    int     down       = right - 1; // rechte Grenze (ohne Trennelement)
    char    t          = a[right];  // rechtes Element als Trennelement
    boolean allChecked = false;     // austauschen beendet
    do {
        while (a[up] < t) {
            up++;              // suche grösseres (>=) Element von links an
        }
        while ((a[down] >= t) && (down > up)) {
            down--;            // suche kleineres(<) Element von rechts an
        }
        if (up < down) {           // Solange keine Überschneidung
            exchange(a, up, down); // austauschen
            up++; down--;          // linke und rechte Grenze verschieben
        }
        else {
            allChecked = true;     // Abbruch
        }
    } while (!allChecked);
    exchange(a, up, right);  // Trennelement an endgültige Position (a[up])
    if (left < up - 1)
        quickSort(a, left, up - 1);  // linke Hälfte
    if (up + 1 < right)
        quickSort(a, up + 1, right); // rechte Hälfte (ohne Trennelement)
}
```

Sie haben einen Überblick über Bibliotheksklassen von Java zum Sortieren.

Sie können einen einfachen endlichen Automaten verstehen, aufzeichnen und in Java umsetzen.

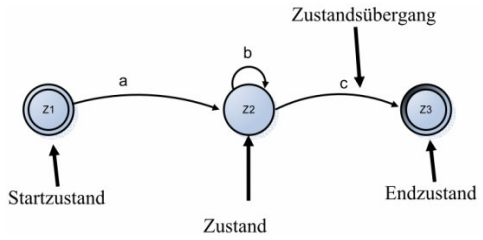
Zwei Klassen enthalten Sortieralgorithmen

- **java.util.Arrays** --> Sortieren von Array
- **java.util.Collections** --> Sortieren von Listen

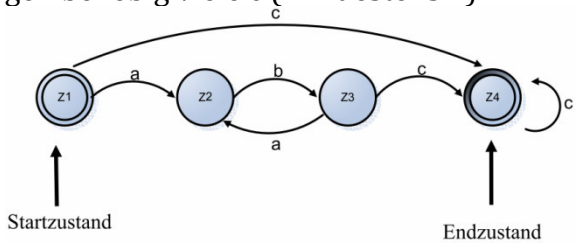
zwei Sortierverfahren werden verwendet

- modifizierter Mergesort (in Klasse Collection) --> stabil
- tuned Quicksort (mit Optimierung)(in Klasse Array) --> instabil

Kaugummiautomaten sind **Automaten,** die bei Eingabe einer Münze einen Kaugummi herausgeben. In der Informatik bezeichnet man als Automaten vorwiegend mathematische Modelle (...), die Zeichenfolgen verarbeiten und dabei Antworten geben, wobei die Eingabe oft nur gelesen aber nicht verändert werden darf.
Beispiel:
Der folgende endliche Automat akzeptiert alle Werte von folgender Struktur:
der erste Buchstabe ist ein a, danach folgen beliebig viele (auch 0-mal) b, danach folgt ein c



Der folgende endliche Automat akzeptiert alle Worte von folgender Struktur:
zu Beginn kommt eine beliebig lange Folge der Zeichenkette «ab» (auch 0-mal), danach folgen beliebig viele c (mindestens 1)



Sie können die Ränder aller Teilworte eines Wortes bestimmen.

Sie können einen "Musterautomaten" zu einem einfachen Muster aufzeichnen.

Der Rand eines Wortes ist die (maximale) Buchstabenfolge, die kürzer ist als das Wort selber und die sowohl am Anfang als auch am Ende des Wortes auftritt.

Ränder aller Teilworte des Musters „abababaa“

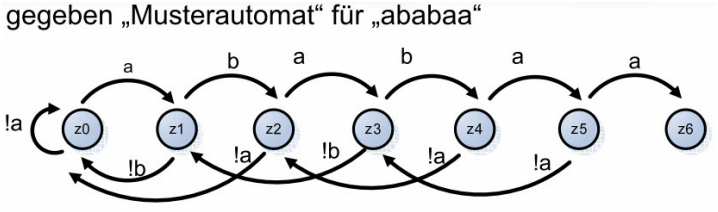
Teilwort	Rand	Länge des Randes
a	Ø	0
ab	Ø	0
aba	a	1
abab	ab	2
ababa	aba	3

Algorithmus von einem **Musterautomaten**

- Man überprüft jedes Zeichen des Textes genau einmal.
- Wenn das Zeichen dem nächsten Zeichen des Musters entspricht, folgt man dem Zustandsübergang gemäss «Musterautomaten»
- Wenn das Zeichen nicht dem nächsten Zeichen des Musters entspricht, folgt man dem Zustandsübergang (!Zeichen) des «Musterautomaten»
- Dies macht man so lange, bis man entweder am Anfangszustand (Z0) ist, oder bis der Wert des Zeichens einem Übergang entspricht, dem man dann folgt.
- Danach überprüft man das nächste Zeichen

Beispiel: oder Automat von Übungsstunde oder im Internet nach Bsp suchen

gegeben „Musterautomat“ für „abababaa“



Sie kennen den Aufwand und die Merkmale des **KMP Suchalgorithmus, von Quicksearch** und von Optimal Mismatch.

Sie kennen den Aufwand und die Merkmale des KMP Suchalgorithmus, von Quicksearch und von **Optimal Mismatch**.

KMP-Algorithmus (Knuth, Morris, Pratt)

- Der KMP-Algorithmus beinhaltet 2 Schritte: Entsprechend dem Pattern (Länge m) den Automaten generieren --> Array next(); Mit den Automaten bzw. mit Hilfe von next die Zeichenkette (Länge n) nach dem Pattern durchsuchen
- Für den KMP-Algorithmus resultiert ein Aufwand von O(m + n), weil sich das Pattern und die Zeichenkette je rein sequentiell Zeichen für Zeichen abarbeiten lassen.

Quicksearch

- Quicksearch ermöglicht schnelles Durchsuchen einer Zeichenkette a,nach einem Text-Pattern.
- Ansatz: Durch die zu durchsuchende Zeichenkette nicht «kriechen» sondern «springen».
- Bei Zeichenunstimmigkeit (Mismatch) zwischen dem Pattern und der Zeichenkette sofort das Zeichen unmittelbar nach dem Pattern analysieren.
- min. tendiert gegen n/(m + 1)

Optimal-Mismatch

- Optimal-Mismatch basiert auf Quicksearch
- Quicksearch besitzt die Eigenschaft, dass Zeichenvergleiche grundsätzlich in beliebiger Reihenfolge stattfinden können.
- Die Optimierung basiert auf der Tatsache, dass die Zeichenhäufigkeit in den meisten Texten nicht gleich verteil ist. o z.B. im Deutschen: Buchstabe e an erster Stelle mit 12 %, Buchstabe x an letzter Stelle mit 0.002%
- Für das Vergleichen zieht man deshalb zuerst die seltenen Zeichen aus dem Pattern heran, so dass die Wahrscheinlichkeit für einem Mismatch bzw. für das Weiterspringen gross ist

--> Optimal-Mismatch

Oft fehlen Statistiken zur Zeichenhäufigkeit. Dann sind andere Strategien gefragt:

- Man reorganisiert das Pattern nach einem Mismatch so, dass das entsprechende Zeichen zukünftig als erstes verglichen wird. Die anderen Zeichen verschieben sich dann einfach um eine Stelle nach hinten.
- Noch einfacher ist es, dieses Zeichen im Pattern nur um eine Stelle nach vorn zu vertauschen.

Optimal-Mismatch besitzt theoretisch die gleichen Grenzen wie Quicksearch für die minimale und maximale Anzahl Vergleiche. Praktisch liefert Optimal-Mismatch aber noch schnellere Suchergebnisse.

Sie können die fünf Hauptdisziplinen des Software Engineering benennen.

Sie sehen die Notwendigkeit des Testens ein und können Gründe dafür benennen.

Kunden-Anforderungen erfassen und beschreiben
System-Spezifikationen

- Anforderungen an Software und Umgebung spezifizieren, Architektur-Übersicht und Datenmodell erstellen, GUI-Skizzen entwerfen, Realisierbarkeit prüfen.

Realisierung

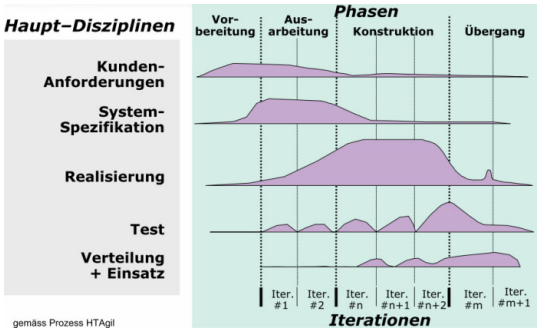
- Software schrittweise entwerfen, programmieren, testen und
- Dokumentieren
- Klassen-Design (vgl. Abstraktion, Interfaces, Abhängigkeiten)
- Programmierung
- Test (vgl. JUnit)
- Dokumentieren (vgl. Javadoc)

Test

- Software schrittweise zusammenführen / integrieren und testen.

Verteilung und Einsatz

- Software verteilen /verfügbar machen und einsetzen.



- Testen = Systematisches Probieren nach verschiedenen Qualitätskriterien.
- Qualität: Übereinstimmung mit (formulierten) Anforderungen, nach Kriterien wie Funktionalität, Zweckdienlichkeit, Robustheit, Zuverlässigkeit, Sicherheit, Effizienz, Benutzbarkeit, Geschwindigkeit
- verschiedene Testmethoden
- Fehlerarten

  Syntaxfehler sind Abweichungen von der Sprachsyntax (formale Fehler)

- Semantische Fehler (inhaltliche Fehler) sind bedeutungsmässige Unstimmigkeiten
- Logikfehler: Ein Programm beschreibt aber den gewünschten Ablauf nicht richtig.
- Qualitätssicherende Begleitmassnahmen: Massnahmen, welche die Softwarequalität positiv beeinflussen, wie z.B. Reviews, definierter Entwicklungsprozess, Walkthrough
- Vertifikation: formaler Nachweis von Eigenschaften von Programmen oder Programmteilen.
- Aussagekraft von Tests:

  Nach einer erfolgreichen Vertifikation weiss man, dass ein Programm für alle möglichen Eingabedaten korrekt ist.

  Nach einem Test weiss man nur, dass ein Programm für die getesteten Eingabedaten korrekt ist. Man möchte aber auf alle möglichen Eingabedaten schliessen.

  Man kann immer nur das Vorhandensein von Fehlern zeigen; nie die Abwesenheit von Fehlern.

  Testmengen müssen typisch und untypische Eingabewerte enthalten (z.B. 1, 2, 3, 4, 5 für ein Sortierprogramm, d.h. bereits sortiert)

  Man bilde Gruppen von Eingabewerten, die gemeinsame Merkmale haben.

Sie kennen die Haupteigenschaften von Blackbox-Test, Whitebox-Test, Walkthrough und Debugger-Einsatz.

Sie können aufgrund einer einfachen Programmspezifikation eine Testspezifikation erstellen.

Sie können Ihren Code gemäss den Richtlinien von javadoc unter Verwendung von javadoc Tags kommentieren.

Sie können die drei Vorgehensschritte beim Refactoring beschreiben.

Black-Box:
Code nicht sichtbar/bekannt, sieht nur In und Output, wird nur betrachtet ob Programm den Testfall erfüllt oder nicht, dabei werden die Testfälle in drei Typen unterteil: Normalfälle, zulässige Spezialfälle, unzulässige Spezialfälle

White-Box:
Code bekannt, Quellcode wird auch analysiert
Datenorientiertestesten, Verwendungsüberdeckung (ob alle Variablen benötigt werden) Wertebereichsüberdeckung (jede definierte Variabel werden die Randwerte getestet)
Code Abdeckung: so programmiert dass alle Pfade mind einmal gebraucht werden

Walkthrough:
Wird der Code von Hand von einer fremden Person durchgegangen

Debugger:
Test zur Laufzeit (Stoppen und aktueller Wert betrachten), ermöglicht ein Step by Step oder Breakpoints setzen, sowie beobachtung der Variablen

Testspezifikation
Beschreibt die im Testplan genannten Vorgehensweisen im Detail

- Unterstützt Programmierer bei der Dokumentation ihrer Klasse
- für die Dokumentation des nach aussen sichtbaren «Interfaces» (was), nicht der Implementation (wie)
- Das Javadoc Tool verarbeitet die Deklarationen und javadoc Kommentare (/**...*/) in Java Source Code Dateien und erzeugt eine Menge von html-Dateien. Damit kann eine Programm-Dokumentation erzeugt werden.

```
/**
* Dieser Text wird angezeigt bis zum ersten Punkt.
*@param
*@return
*@author
*@version
*/
```

Refactoring, Anpassung des Codes/Design an neue Anforderung, gutes Klassen- Desing kann nie alle Eventualitäten berücksichtigen und muss bei Änderungen getestet und angepasst werden um Fehler zu vermeiden.

Vorgehen:
- Falls der bestehende Code noch nicht getestet wurde: Testspezifikationen schreiben und den Code testen.
- Code redesignen ohne funktionale Änderungen und Code erneut testen (analog oben).
- Code erweitern mit funktionalen Änderungen und Code erneut entsprechend testen.