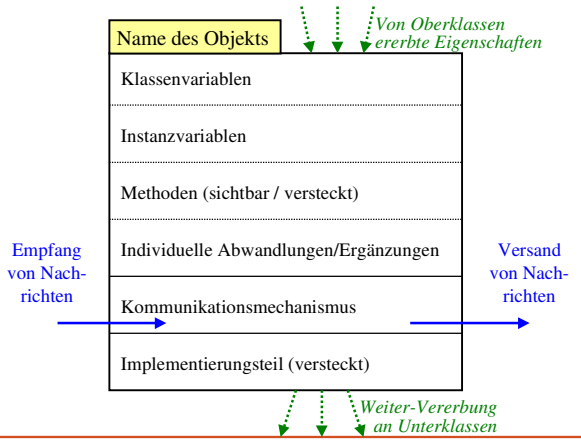
Gliederung des Kapitels 1.4

1.4/1.5 Programmierung (und die Sprache Ada 95)

~~1.4.1 Blöcke, Deklarationen und Ausnahmen~~
~~1.4.2 Prozeduren und Funktionen~~
~~1.4.3 Moduln~~
~~1.4.4 Polymorphie~~
~~1.4.5 Vererbung~~
~~1.4.6 Abstrakte und konkrete Datentypen~~
1.4.7 Objekte
1.4.8 Grundprinzipien, Paradigmen der Programmierung

1.4.7 Objekte (etwas ausführlicher: siehe Vorlesung)

1.4.7.1: Objekte sind in sich geschossene Einheiten, die
- wie Moduln aufgebaut sind: Es gibt ein Schema, genannt "Klasse", das vor allem aus "Attributen" (das sind die einzelnen Datenstrukturen der Variablen) und "Methoden" (das sind die algorithmischen Teile) einschl. der Angaben zur Sichtbarkeit besteht und aus dem ein neues Objekt erzeugt werden kann; das Objekt ist eine Instanz (oder ein "Exemplar" oder eine "Ausprägung") dieser Klasse.
- einen individuellen Zustand besitzen (Speicherzustand der Klassen- und Instanzvariablen),
- miteinander kommunizieren können; dies geschieht durch Nachrichtenaustausch ("message passing"),
- durch Vererbung ihre Eigenschaften an neue Objekte bzw. Klassen weitergeben können.



1.4.7.2 Prinzipien der Objektorientierung:

1. Es gibt nur Objekte. Jedes Objekt ist eindeutig identifizierbar über seinen Namen.
2. Alles wird über Klassen, Instanzbildung, Zustände, Methoden, Nachrichten und Vererbung realisiert.
3. Objekte handeln in eigener Verantwortung (und sie geben nur bekannt, *was* sie bearbeiten, niemals, *wie* sie dies tun).
4. Klassen werden in Bibliotheken aufbewahrt und stehen allen Programmen und Klassendefinitionen zur Verfügung.
5. Programmieren bedeutet, Klassen festzulegen, hieraus Objekte zu erzeugen und diesen Aufgaben zu übertragen, indem man ihnen geeignete Nachrichten schickt. Die Auswertung der Objekte erfolgt hierbei erst zur Laufzeit (Polymorphie, dynamische Bindung der Objekte an Variable).

Wenn "alles" Objekte sind, so sind konsequenterweise auch Klassen, Nachrichten und die (formalen) Parameter Objekte.

Eine Methode der Klasse *"Klasse"* ist "new". Diese Methode erzeugt aus der Klasse eine Instanz, also ein konkretes Objekt. Jede Klasse ist eine Unterklasse der Klasse *"Klasse"* und hat somit diese Methode ererbt, kann also Instanzen von sich selbst erzeugen.

Eine konkrete Nachricht, die ein Objekt A an ein Objekt B schickt, besitzt meist aktuelle Parameter. Diese sind ebenfalls Objekte. Ebenso erwartet das Objekt A, dass das Objekt B ihm eine Nachricht mit konkreten Objekten als aktuellen Parametern zurückschickt.

1.4.7.3 Hinweise (1):

a. Klassen bilden *Hierarchien* oder zyklenfreie Abhängigkeitsgraphen (Ober- / Unterklassen, einfache / mehrfache Vererbung).
b. In typisierten Sprachen verweist eine Variable auf ein Objekt ihrer Klasse oder einer ihrer Oberklassen. Anderenfalls muss die Zuordnung laufend auf ihre *Zulässigkeit* überprüft werden.
c. Ist das zu aktivierende Objekt identifiziert, so muss man ermitteln, *welches die angeforderte Methode konkret ist* (eventuell muss man diverse Oberklassen durchsuchen) und ob es sie ausführen kann.
d. Zu jeder objektorientierten Sprache gibt es umfangreiche *Klassenbibliotheken*, aus denen man sich sein Programm aufbauen kann.
e. In der Praxis benötigt man eine *Entwurfsumgebung*, um Programme im Team zu entwickeln, um Erläuterungen, Entwurfsprozesse und verschiedene Dokumentationen zu erstellen und um die Klassenbibliothek zu erweitern.

Hinweise (2): Probleme in der Praxis, kleine Auswahl:

f. Die Sprache muss Erweiterungsmöglichkeiten von Moduln haben, ohne dass hierbei die privaten Informationen bekannt werden. (Das ist oft nicht realisierbar. Ada: child library units.)

g. Es müssen verschiedene Sichtweisen auf ein Objekt möglich sein. (Hierfür kann man Mehrfachvererbung nutzen.)

h. Es müssen Teile getrennt voneinander übersetzbar sein und abgelegt werden, allerdings darf hierdurch die Sichtbarkeit nicht beeinträchtigt werden. (Stichwörter in Ada: separate, stub.)

i. Die Klassenbibliotheken sollten sowohl den Quellcode als auch bereits getrennt compilierte Teile besitzen, und zwar so, dass diese leicht in ein Programm (z.B. über with und use) eingefügt werden können.

Hinweise (3): Probleme in der Praxis, kleine Auswahl:

j. Die Eindeutigkeit von Namen ist zu gewährleisten, z.B. durch Umbenennung importierter Größen (in Ada renames:
with Stack_für_Zeichen;
X: StackZ renames Stack_für_Zeichen.S;
function "*" (X,Y: Vektor) return float renames Skalarprodukt;)

k. Die Sichtbarkeit in der Vererbungshierarchie ist genau festzulegen. (Beispiel: In der Regel sehen Unterklassen nicht, wie ihre Oberklassen die Methoden realisiert haben; daher können sie diese auch nicht verwenden, um Umdefinitionen vorzunehmen. Ändert man eine Methode ab, so muss man aber meist Zugriff auf jene soeben ausgeblendete Methode haben. In Ada: über Punkt-Notation. In Java durch "super".)

l. Die Sichtbarkeitsregeln müssen auch in der Klassenbibliothek gelten. Beispielsweise wird durch "with" die Sichtbarkeit einer anderen Klasse importiert (wann und wo endet diese?).

Hinweise (4): Probleme in der Praxis, kleine Auswahl:

m. Wie entwirft man "objektorientiert"? Man unterscheidet zwischen OOA und OOD, also "objektorientierte Analyse" und "objektorientierter Entwurf" ("D" = design = Entwurf). In der OOA wird ein Sollkonzept/Pflichtenheft erstellt und für dieses werden geeignete Klassen mit Zusatzinformationen (zeitliche Abläufe, einzuhaltende Bedingungen, Zusammenwirken der Einheiten, ...) erarbeitet. Im OOD werden hieraus die tatsächlichen Klassen, möglichst aus einer Klassenbibliothek und ergänzt um zusätzlich erforderliche Hilfs-Klassen erstellt und die Realisierung in einer Programmiersprache skizziert. (Anschließend folgt die Codierung.)

n. Die Zeit, dass man Lösungen zu Problemen von Grund auf neu schrieb, ist vorbei. Heute versucht man, eine Problemlösung so zu beschreiben, dass sie mit Hilfe der vorhandenen Klassen realisiert werden kann. Nur für wenige Probleme werden noch neue Klassen, neue Datentypen, neue Vorgehensweisen entwickelt (die anschließend in die Klassenbibliothek übernommen werden).

1.4.7.4 Beispiel: Addieren von Zahlen

Zur Illustration des Prinzips objektorientierter Denkweise:
Die Addition zweier Zahlen "7 + 28" läuft aus objektorientierter Sicht folgendermaßen ab:

Das Objekt 7 bekommt die Nachricht, es möge seine Methode "+" auf sich und das beigefügte Objekt (aktueller Parameter 28) anwenden und das Ergebnis zurückschicken.

Die Zahl 7 ist eine Instanz der Klasse "integer", die aus einem Zustand (dies ist der Wert 7 seiner Instanzvariablen INH) und aus den zulässigen Methoden "+", "abs", "*", "-" usw. sowie den allgemeinen Methoden "send" (=schicke das Objekt, das auf "send" folgt, an den Sender zurück), "write" (drucke den Wert von INH aus) usw. besteht.

Wir haben also eine Klasse "integer" (Oberklasse sei "Zahlen")

integer	Vererbt von "Zahlen" werden: Typ "Binärfolge" und hierauf die Operationen plus, minus, mal, "<", "=", wie sie üblich sind. Der Ausbau zu "integer" erfolgt jetzt:
Klassenvariablen	Null: constant Binärfolge := 0; Zehn: constant Binärfolge := 1010
Instanzvariablen	INH: Binärfolge
Methoden	function "+" (X: integer) return integer; function quad return integer; function "abs" return Binärfolge; ...
Kommunikation	procedure return1 (A:Binärfolge) is begin X := new Integer; X.INH := A; send X end; procedure send (A:Object) is begin "schicke A an den Sender zurück" end; ...
Implementierung	procedure "+" (X: integer) is begin return1 (plus (self.INH, X.INH)) end; procedure abs is begin if self.INH<0 then send minus(Null,self.INH) else send self.INH fi end; procedure quad (X: integer) is begin return1 (mal (self.INH, self.INH)) end; ...

Es bleibt der jeweiligen Sprache überlassen, ob die im sichtbaren Bereich aufgelisteten Funktionen tatsächlich als Funktionen oder auf andere Weise (z.B. wie hier als Prozeduren; aber durch "return1" wird die richtige Sicht nach außen hergestellt) implementiert werden.

Z := 7 + 28 wird daher wie folgt ausgerechnet
(die beiden Zahlen werden als Binärfolge dargestellt):

X := new integer; X.INH := 111;

Y := new integer; Y.INH := 11100;

Z := X."+"(Y);

X."+"(Y) bedeutet: Schicke an X die Nachricht, dass dessen Operation "+" ausgeführt werden solle. Diese Funktion erwartet ein Objekt der Klasse integer als Parameter, daher wird Y als aktueller Parameter mitgegeben. Das Objekt X antwortet dann mit einem Objekt der Klasse integer, das an die Variable Z gebunden wird.

1.4.7.5 Beispiel: Geometrische Größen

Geometrische Größen entwickelt man gerne auseinander. Man beginnt mit einem Punkt, geht zur Strecke und zum Polygon über, dann betrachtete man Dreieck, Viereck und n-Ecke, führt Kreise (= Punkt plus Strecke) und Ellipsen ein usw.

Zugleich kann man schrittweise weitere Eigenschaften hinzufügen (insbesondere die Fläche) und Unterklassen bilden (z.B.: Viereck → Trapez → Raute → Quadrat), wobei zugleich die Flächenberechnung jeweils neu definiert werden kann.

Die Darstellung verändern wir nun etwas, indem wir die Objekte wie Moduln ausformulieren und die Oberklasse(n) explizit angeben.

Wir definieren also eine Klasse "Punkt" (Oberklasse sei real):

Punkt	real
X: real; Y: real;	
procedure Drucken	
procedure Drucken is begin < sende an das Objekt *Drucker* die Nachricht drucke_ein_Pixel_an_die_Position (self.X, self.Y) > end	

Wir definieren nun eine Klasse "Strecke":

Strecke	Punkt, real
Anfang: Punkt; Ende: Punkt;	
procedure Drucken, DruckeAnfang, DruckeEnde; function Länge ( ) return real	
procedure Drucken is begin < sende an das Objekt *Drucker* die Nachricht drucke_eine_ Strecke_von_bis (self.Anfang.X, self.Anfang.Y, self.Ende.X, self.Ende.Y) > end; procedure DruckeAnfang is begin Anfang.Drucken end; procedure DruckeEnde is begin Ende.Drucken end; function Länge ( ) return real is begin return (square_root(quad(self.Anfang.X - self.Ende.X) + quad(self.Anfang.Y - self.Ende.Y))) end;	

Wir definieren nun eine Klasse "Polygon" mit dem generischen Parameter N (Oberklassen sind Strecke und integer):

Polygon (N: natural, N > 2)	Strecke, natural
Eckpunkte: array [1..N] of Punkt; private Streckenzug: array [1..N] of Strecke;	
procedure Drucken; function Länge ( ) return real	
procedure Drucken is var I: natural; begin for I := 1 to N do Streckenzug[I].Drucken od end; function Länge ( ) return real is var I: natural; L: real := 0.0; begin for I := 1 to N do L := L + Streckenzug[I].Länge od; return L end;	

var I, K: natural;
begin for I := 1 to N do Streckenzug[I] := new Strecke;
Streckenzug[I].Anfang := Eckpunkte[I];
if I = N then K:=1 else K:=I+1 fi;
Streckenzug[I].Ende := Eckpunkte[K] od
end
	Initialisierung

Beachten Sie hierbei:

Mit der Klasse "Strecke" ist auch deren Oberklasse "Punkt" eine Oberklasse von Polygon. Daher können wir deren Eigenschaften hier verwenden.

Am Ende haben wir einen Initialisierungsteil angefügt, mit dem wir die Folge der Strecken, die durch die Eckpunkte definiert ist, errechnen. Diese Information geben wir nach außen nicht bekannt ("private"). Den Streckenzug verwenden wir intern zum Drucken und zur Längenberechnung.

Sie erkennen hier einen Vorteil der OOP (= objektorientierten Programmierung): Wenn wir die Klasse "Punkt" von "real" auf "integer" umschreiben würden, so braucht an Strecke und Polygon nichts geändert zu werden! (Wiederverwendbarer Quellcode.)

Nun müsste man eigentlich noch

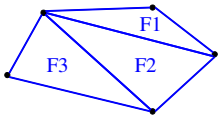
- eine Boolesche Funktion "kreuzungsfrei" hinzufügen, die genau dann den Wert true ergibt, wenn sich je zwei Strecken des Polygons nicht schneiden. Hierzu würde man in der Klasse "Strecke" eine Methode einfügen:

  function schnitt (S: Strecke) return Boolean is
  var ...
  begin ... end;

  (Selbst ausformulieren; Erläuterungen auf später folgenden Folien.)
- eine Boolesche Funktion "konvex" hinzufügen, die genau dann den Wert true ergibt, wenn das Polygon kreuzungsfrei ist (dann hat es ein "Inneres") und wenn alle Winkel zum Inneren des Polygons hin höchstens 180 Grad betragen.

Nun müsste man eigentlich noch

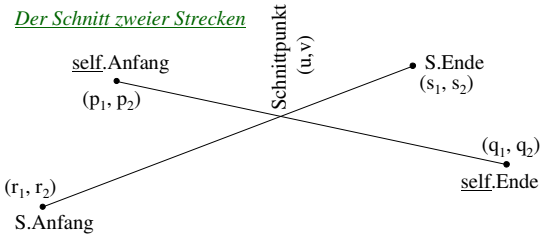
- für konvexe Polygone eine "Fläche" hinzufügen. Diese erhält man, indem man von einem Punkt aus alle aufeinander folgenden Dreiecksflächen aufsummiert (die Dreiecksfläche ist durch drei Seitenlängen eindeutig bestimmt und hieraus leicht zu berechnen); Beispiel:



Gesamtfläche = F1 + F2 + F3.

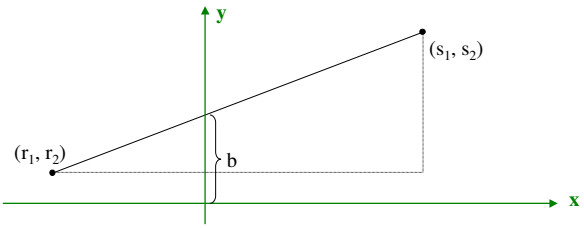
- die Flächen ausweiten auf kreuzungsfreie Polygone.
- den Schwerpunkt eines Polygons bestimmen oder andere "Mittelpunkte".

Der Schnitt zweier Strecken



Den Schnittpunkt erhält man als Schnittpunkt der beiden Geraden, die zu den Strecken gehören. Für zwei Geraden $y = a_1 \cdot x + b_1$ und $y = a_2 \cdot x + b_2$ ergibt sich der Schnittpunkt (u,v) durch $v = a_1 \cdot u + b_1$ und $v = a_2 \cdot u + b_2$, also für $a_1 \neq a_2$:

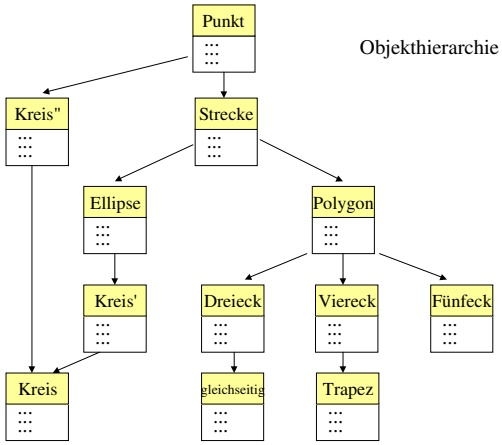
$$u = \frac{b_2 - b_1}{a_1 - a_2} \quad \text{und} \quad v = a_2 \cdot \frac{b_2 - b_1}{a_1 - a_2} + b_2$$



Ermittlung der Steigung a und des y-Abschnitts b der zu einer Strecke gehörenden Geraden $y = a \cdot x + b$:

$$a = \frac{s_2 - r_2}{s_1 - r_1} \quad \text{und} \quad b = s_2 - a \cdot s_1$$

Ein Schnitt liegt vor, wenn die x- oder die y-Koordinate des Schnittpunkts zwischen den entsprechenden Koordinaten einer der beiden gegebenen Strecken liegt. Rest: selbst ausführen.



1.4.7.6 Beispiel: Skat spielen

Kurzanalyse: Drei Spieler, die alle die Spielregeln kennen, die reizen, ausspielen und beilegen können, die wissen, wer den jeweiligen Stich gewonnen hat, die am Ende jedes Spiels den Sieger des Spiels einschl. der Punktzahl ermitteln und dies in eine geeignete Tabelle eintragen können. Am Ende entnimmt man der Tabelle den Gesamtsieger bzw. die individuellen Spielschulden jedes Spielers.

Es wird nur angedeutet, wie der Verlauf des Skat-Spiels modelliert werden könnte. In der Praxis würde man zunächst eine Klasse "Kartenspiel" mit diversen Methoden (Auflisten der Karten, Mischen, Verteilen nach verschiedenen Vorgaben usw.) einführen, ebenfalls eine Spiel-Verwaltung (für Stand, Punkte, abgelegte Karten usw. Das Folgende soll daher nur "anregen".

Bekannt seien aus Oberklassen: type Karte und das "array" Kartenspiel in Form von "alle_karten"

Tisch

```
subtype bis_drei is natural range 0..3;  subtype nr is bis_drei range 1..3;
Skat: array [1..2] of Karte; Blatt: array [nr, 1..10] of Karte;
Anz_gesp_Karten: bis_drei;  gespielte_Karten: array [nr] of Karte;
Anzahl_Spiele: natural;  Punktestand: array [nr] of integer;
Erster_Spieler: nr;  Ausspielend, Abhebend, Geber, Gewinner: nr;
Aktuelles_Spiel: Spiel_Typen; gereizt_bis: natural; Alleinspieler: nr;
Abgelegt: array [Karte] of Boolean; constant KS := alle_karten;
```

```
procedure Karten_Verteilen; procedure reizen (S,T: Spieler);
procedure Ein_Stich; procedure Spiel_Ergebnisse;
function vor (K: nr) return nr;  function nach (K: nr) return nr;
procedure End_Abrechnung;
```

```
function vor (K: nr) return nr is
begin if K = 1 then return 3 elsif K = 2 then return 1 else return 2 fi end;
procedure Karten_Verteilen is var I: nr; M: array[1..32] of Karte;
begin M := Zufalls_Permutation(KS); Skat := "die ersten beiden Karten";
   for I in nr do Blatt [I,1..10] := "die nächsten 10 Karten" od end;
...
```

Spieler	Bekannt sei die Klasse Tisch
subtype bis_drei is natural range 0..3; subtype nr is bis_drei range 1..3; Skat: array [1..2] of Karte; Blatt: array [1..10] of Karte; Anz_gewonn_karten: 0..32; gewonnene_Karten: array [1..32] of Karte; Aktuelles_Spiel: Spiel_Typen; gereizt_bis: natural; Alleinspieler: nr; gewonn_Punkte_bisher: natural; Abgelegt: array [Karte] of Boolean; Reaktionen_der_anderen: <neuen Datentyp einführen ...für Strategie>; ich: nr; Reiz_Obergrenze: natural; mögliches_Spiel: Spieltypen; Spielverlauf: list of ...; Reiz_Verlauf: list of ...;	
procedure Karten_Sortieren; procedure reizen; function Ausspielen return Karte; function Beilegen return Karte; procedure Spielweisen_analysieren;	
function Ausspielen return Karte is var ...; begin ... return := ... end; procedure Blatt_bewerten is var ...; begin ... end; procedure reizen is var ...; begin ... end; ...	

Ein Programm muss dann den Ablauf in geordneter Weise steuern, also etwa:

```
... "Initialisiere den Tisch und die Spieler;"
while noch_nicht_Schluss do
  Tisch.Karten_verteilen; "Initialisiere die Spieler";
  Tisch.reizen (Abhebend,Ausspielend);
  Tisch.reizen (Geber, Gewinner);
  "Initialisieren aller Spieler und des Tisches für das Spiel";
  for I := 1 to 10 do Ein_Stich od;
  Tisch.Spiel_Ergebnisse
od;
Tisch.End_Abrechnung; ...
```

Wollen Sie es selbst einmal weiter durchdenken, allgemein formulieren und dann in Ada programmieren? Geschätzter Zeitaufwand bis zu einer lauffähigen ersten Version: 40 Stunden (ohne Graphik).