# Alan Kay on formuleerinud

## **OOP 5 tähtsaimat printsiipi**

1. **Everything is an object**. Iga objekt on omapärane muutuja – objekt (nagu iga muutuja) hoiab endas **andmeid**, mis määravad objekti oleku. Iga objektiga on aga lahutamatult seotud ka mingid funktsioonid (**meetodid**), mille kaudu saab objekti olekut (objekti käitumist!) muuta. Programmis võib objektiks kuulutada praktiliselt suvalise mõiste.
2. **Programm on objektide kogum, mis saadavad üksteisele sõnumeid**. Kui üks objekt soovib saata mõnele teisele objektile sõnumi (näit. päringu tema oleku kohta), siis see objekt pöördub teise objekti vastava meetodi (funktsiooni) poole.
3. **Igal objektil on oma isklik mälu, mis on reeglina suletud teistele objektidele**. Seetõttu jääb objekti ehitus varjatuks *(hidden implementation*), sest objektiga saab reeglina suhelda ainult tema meetodite (objekti liidese, *interface*) kaudu.
4. **Iga objekt on konkreetset tüüpi**. Ehk teisiti – iga konkreetne objekt on mingi objektitüübi (**klassi**) ilming (*instance*). Klassi kasutaja tahab teada vastuseid küsimustele:

* mis teenuseid seda tüüpi objektid pakuvad?
* milliseid sõnumeid saab seda tüüpi objektile saata?

1. **Kõik sama tüüpi objektid on ühesuguse ehitusega**. Seetõttu kõik antud tüüpi objektid oskavad vastu võtta ühesuguseid sõnumeid.

**Üldisematest objektitüüpidest on võimalik tuletada (*derive*) alamtüüpe**. Näiteks kui on loodud klass Kujund, siis on võimalik sellest objektitüübist tuletada objektitüüp Ring, mis **pärib** (*inherite*) kõik klassi Kujund omadused (andmed ja meetodid). Seetõttu oskab iga Ring-tüüpi objekt vastu võtta ka kõiki selliseid sõnumeid, mida oskavad käsitleda Kujund-tüüpi objektid.

Seega – kirjutades üks kord valmis klassi Kujund programmikoodi (meetodid), kandub see programmilõik automaatselt üle kõikide klassist Kujund tuletatud klasside programmikoodi!

Tuletamise-pärimise skeem ongi kõige tähtsam mehhanism, mis tagab OOP võimsuse ja efektiivsuse.

#### OOP KEELED

**Kolm omadust**:

1. **kapseldamine** (*encapsulation*) – keeles on vahendid, mis võimaldavad koondada andmed ja nendega tehtavad operatsioonid (meetodid) ühte programmilisse ühikusse (objektitüüp)

# NB! objekti andmetele saab reeglina ligi ainult meetodite

kaudu!

1. **pärandamine** (*inheritance*) – igast objektitüübist saab tuletada uusi objektitüüpe, kusjuures järglased pärivad eellase omadused (nii andmed kui meetodid)
2. **polümorfism** (*polymorphism*) – samanimelised meetodid võivad erinevatel objektidel teostuda erineval viisil

### Objektitüüp Stack

pop

## view

push

init

OP 1

OP 2

## OP 4

OP 3

#### Objektitüübid keeles C++

Kirjeldajad: **struct union class**

<kirjeldaja> <objektitüübi\_identifikaator>

{

<muutujad\_ja\_meetodid\_=\_LIIKMED\_(*members*)>

}

// IntStack.cpp - täisarvude LiFo

#include <iostream.h>

struct **Elem**

{

int **arv**;

Elem \***next**;

};

class **Stack**

{

private:

Elem \***top**;

public:

void **init**( void ) { top = NULL; }

void **push**( int a )

{

Elem \*uus = new Elem;

uus -> arv = a;

uus -> next = top;

top = uus;

}

int pop( int &a)

{

if( !top ) return 0; // FALSE kui tühi

Elem \*vana = top;

a = top -> arv;

top = top -> next;

delete vana; // vabastada mälu

return 1; // TRUE kui võetakse element

}

void view( void )

{

Elem \*jooksev = top;

int n = 0;

while( jooksev )

{

cout << jooksev -> arv << "\n";

n++;

};

cout << "\n Kokku " << n <<

" elementi\n";

}

};

int **main**( void )

{

Stack mag; // uus ilming

mag.init();

for( int i=0; i<10; i++)

mag.push( i );

mag.view();

Stack pinu; // uus ilming

pinu.init();

int num;

while( mag.pop( num ))

pinu.push( num );

pinu.view();

pinu.top ->arv = 7; // Viga! privaatne!

mag.top = NULL; // Viga! privaatne!

return 0;

}

Juurdepääsu määrajad: **public private protected**

1. klassi (objektitüübi) meetodites on juurdepääs **kõigile** selle klassi liikmetele
2. objektitüübi ilmingu kaudu on juurdepääs (liitnime abil) **ainult avalikele** liikmetele

Vaikimisi on liikmed:

class -> private

struct -> public

union -> public

**Konstruktorid ja destruktorid**

Konstruktor:

* liikmefunktsioon, mille nimi ühtib klassi nimega
* konstrutoril puudub väärtus (ka mitte *void !*)
* konstruktor käivitatakse objekti ilmingu deklareerimise hetkel (ja ainult siis! – liitnimega ei saa pöörduda!)
* konstruktoril võivad olla parameetrid

Näide:

muudame klassi **Stack** meetodi **init** konstruktoriks:

public:

**Stack** ( void ) { top = NULL; }

// põhiprogrammis nüüd niimoodi:

Stack mag; // käivitub konstruktor!

Destruktorid:

* liikmefunktsioonid objekti ilmingu “hävitamiseks”
* nimi nagu konstrutoril, aga ees **~**  (tilde)
* lubatud väljakutse liitnimega
* objekti ilmingule rakendub destruktor automaatselt, kui programm “väljub” plokist, kus see ilming on deklareeritud

Näide: klassi **Stack** destruktor:

public:

~Stack ( void )

{

int b;

while( pop( b )) ;

}

// nüüd võib põhiprogrammis teha nii:

mag.~Stack();

#### Klassi prototüüp

Näide - klassi Stack prototüüp:

class Stack

{

Elem \*top;

public:

Stack( void );

void push( int );

int pop( int & );

void view( void );

~Stack( void );

}

// Kui moodulis on kirjeldatud klassi prototüüp, siis selle klassi

// liikmefunktsioonide (meetodite) definitsioonides tuleb

// funktsiooni päises ära näidata klassi nimi!

Stack :: Stack( void ) { top=NULL; }

void Stack :: push( int a ) {

// meetodi keha

}

***inline* liikmefunktsioonid**

Liikmefunktsiooni (meetodi) definitsioon võib paikneda:

1. klassi deklaratsiooni “sees”
2. väljaspool klassi deklaratsiooni (pärast klassi prototüüpi)

Juhul 1 loetakse meetod **inline**-liikmefunktsiooniks

Juhul 2 muutub meetod **inline**-liikmefunktsiooniks ainult siis, kui meetodi deklaratsioon algab võtmesõnaga **inline**

Erinevus kajastub meetodi poole pöördumise realisatsioonis:

if *inline*

then *makrolaiendus*

else *call*  <meetod>

Näide:

**inline** void Stack :: **pop**( int &a)

{

Elem \*vana = **this->top**;

**a** = **this->top** -> arv;

**this->top** = **this->top** -> next;

delete vana;

return; // vaikimisi

}

// *inline*-funktsiooni võib nimetada ka **makroks** (makrokirjelduseks),

// sest sellise funktsiooni (makro) väljakutse - näiteks

mag.pop( num ) // tavaliselt vastab sellele *call*-lause !!!

// asendatakse kompilaatori poolt umbes sellise lausete jadaga

// (ehk **makrolaiendusega**):

Elem \*vana = **mag.top**;

**num** = **mag.top** -> arv;

**mag.top** = **mag.top** -> next;

delete vana;

// formaalsed parameetrid on selles lõigus asendatud faktiliste

// parameetritega, sealhulgas *this->member* on asendatud

// *mag.member* –iga;

// samuti pole enam vajadust käskude *call* ja *return* järele

Võidame – töökiiruses

Kaotame – mälus

Millal on mõistlik kasutada inline-funktsioone (makrosid)?

Kirjeldajat *inline* võib kasutada suvalise C++ funktsiooni päises!**Viidad objektidele**

Stack mag, \*p; // p on viitmuutuja Stack-tüüpi objektile

p = &mag; // nüüd p viitab objektile mag

p -> push( 17 );

Stack tosin[12]; // 12 Stack’i

tosin[4].push( 999 );

p = &tosin[7];

p = tosin; // sama mis p = &tosin[0];

**this**

Võtmesõna **this** on klassi meetodites predefineeritud viitmuutja, mille väärtuseks on viit “jooksvale” objektile:

class Ilus

{

int x;

public:

Ilus( void )

{

this -> x = -1; // sama mis x = -1;

}

Ilus \*aadress( void )

{

return this; // !!!

}

}

**Funktsioonide *overload***

Programmis (ka objektitüübis) võib olla **mitu sama nimega funktsiooni** (ka mitu konstruktorit), kui on kas erinev arv parameetreid VÕI parameetritel erinevad tüübid

Näide:

class Midagi

{

public:

Midagi( int ); // 1

Midagi( char \* ); // 2

Midagi( int, int ); // 3

}

// kuskil edasi:

Midagi k( 14, -3 ), // 3

l( 56740 ), // 1

m(“tere”), // 2

n; // Viga!

**Funktsioonide parameetrite vaikeväärtused**:

void proc( int n = 3, char c = ’x’ )

{

}

// ja kuskil mujal:

proc( );

proc( 7 );

proc( , ’Y’ );

proc( -3, 33 );**Objektid ja operaatorite *overload***

Olgu antud objektitüüp:

class Paar

{

int x, y;

public:

Paar( int a, int b) { x = a; y = b; }

Paar( void ) { x = 0; y = 0; }

int getX( void ) { return x; }

int getY( void ) { return y; }

void muuda( int a = 1, int b = 1 ) { x = a; y = b; }

}

Defineerime Paar’ide jaoks:

* liitmise +
* omistamise =
* operaatori ++
* skalaariga korrutamise \*
* skalaarkorrutise \*

Selleks tuleb klassi Paar deklaratsiooni lisada uued meetodid:

Paar operator + ( Paar teine )

{

Paar summa( x + teine.x, y + teine.y );

return summa;

}

Paar operator = ( Paar parem )

{

x = parem.x; y = parem.y;

return \*this; // omistamisel on väärtus!

}

Paar operator ++ ( void )

{

x++ ; y++ ; return \*this;

}

Paar operator \* ( int kordaja )

{

x \*= kordaja; y \*= kordaja; return \*this;

}

int operator \* ( Paar teine )

{

return x \* teine.x + y \* teine.y;

}

}

// kasutamise näide

int main( void )

{

Paar u( 4, -3), v( -2, 5), w;

w = u + v; u++;

v = w \* 2; // skalaariga ainult paremalt!

int s = u \* v;

w.muuda( 7, -6);

u.muuda( -4 ); // ( -4, 1)

v.muuda( , 8 ); // ( 1, 8)

w.muuda( ); // ( 1, 1)

}

Reeglid:

* ei saa muuta operaatorite aarsust
* ei saa muuta operaatorite prioriteete avaldises
* ++v on lubatud, kuid realiseeritakse kui v++

# KLASSI SÕBRAD

class **Sõbralik**

{

friend **semu, Drug**;

private:

int m;

protected:

float x;

}

void **semu**( Sõbralik &kuju )

{

kuju.m = 7; // lubatud !!!

kuju.x = 3.3; // lubatud !!!

}

class Drug

{

public:

Drug( Sõbralik &vend )

{

vend.m = -3; // lubatud !!!

vend.x = 12.34; // lubatud !!!

}

}

# KLASSI STAATILISED LIIKMED

Staatilised liikmed on “ühised” kõigile selle klassi objektidele

class Xklass

{ public:

static int yhine;

static void proge( void ) { ……….. }

}

Xklass a, b;

a.yhine = 3; a.proge(); // ei tohi !!!!!!

Xklass :: yhine = 4; Xklass :: proge(); // OK