Programozási nyelvek és módszerek

7. ELŐADÁS – OOP

Altípus

Szabályok:

- Reflexív
  - $T \subseteq T$
- Tranzitív
  - Ha: $T1 \subseteq T2$; $T2 \subseteq T3$
  - Akkor: $T1 \subseteq T3$

Altípus – alprogramok esetén

Tegyük fel, hogy Triangle $\subseteq$ Shape

- fss = proc (Shape) returns (Shape)
- fts = proc (Triangle) returns (Shape)
- fst = proc (Shape) returns (Triangle)
- ftt = proc (Triangle) returns (Triangle)

- fts $\subseteq$ fss ?
- fst $\subseteq$ fss ?
- ftt $\subseteq$ fss ?
- fss $\subseteq$ fts ?

Hogyan döntsünk?

Egy függvényt A → B alakban írunk, ha A típusú paramétere van és B típusú eredményt ad.

Ha (A’ → B‘) <: (A → B), - altípus

- akkor képesek kell legyünk arra, hogy az első függvénytípus egy elemét használhassuk minden olyan kontextusban, ahol a másodikat.

Tegyük fel, hogy van egy f: A→B típusú függvényünk.

- Ha egy f’ : A’ → B‘ függvényt az f helyén akarunk használni, akkor A-beli argumentumot kell fogadnia és B-beli eredményt adnia.

Hogyan döntsünk?

Mivel az f' értelmezési tartománya A', így akkor alkalmazható A-beli elemekre, ha A<:A', vagyis

- ha a : A tekinthető mint A'-beli, és f'(a) típus-helyes.

Másrészt, ha az f' eredménye B'-beli, akkor B'<:B garantálja, hogy az eredmény B-beliként kezelhető.

Összegezve:
(A' → B') <: (A → B), ha A<:A' és B'<:B

Altípus – Megoldás

fss = proc (Shape) returns (Shape)

fts = proc (Triangle) returns (Shape)

fst = proc (Shape) returns (Triangle)

ftt = proc (Triangle) returns (Triangle)

~~fts ⊆ fss~~

~~ftt ⊆ fss~~

fst ⊆ fss

fss ⊆ fts

Az eredmény lehet speciálisabb
(monoton = kovariáns)

A paraméterek kevésbé speciálisak lehetnek
(anti-monoton = kontravariáns)

Mi az objektumorientált programozás?

A programozó definiálhat altípus kapcsolatokat

A típusszabályok megengedik, hogy az altípus használható legyen a szupertípus helyén (altípusos polimorfizmus)

Típus-vezérelt metódus elérés (dinamikus kötés)

Implementáció megosztása (öröklődés)

Típus-vezérelt metódus elérés

```
s: Shape := new Triangle (3, 4, 5);
s.draw();
```

Statikus elérés:

A Shape draw metódusát hívja

Dinamikus elérés:

A Triangle draw metódusát hívja

Elérési döntések

C++

- Az **őstípus** `virtual`-nak deklarálja a metódust, amire megengedi a felüldefiniálást

Más programozási nyelveknél ez másképp lehet

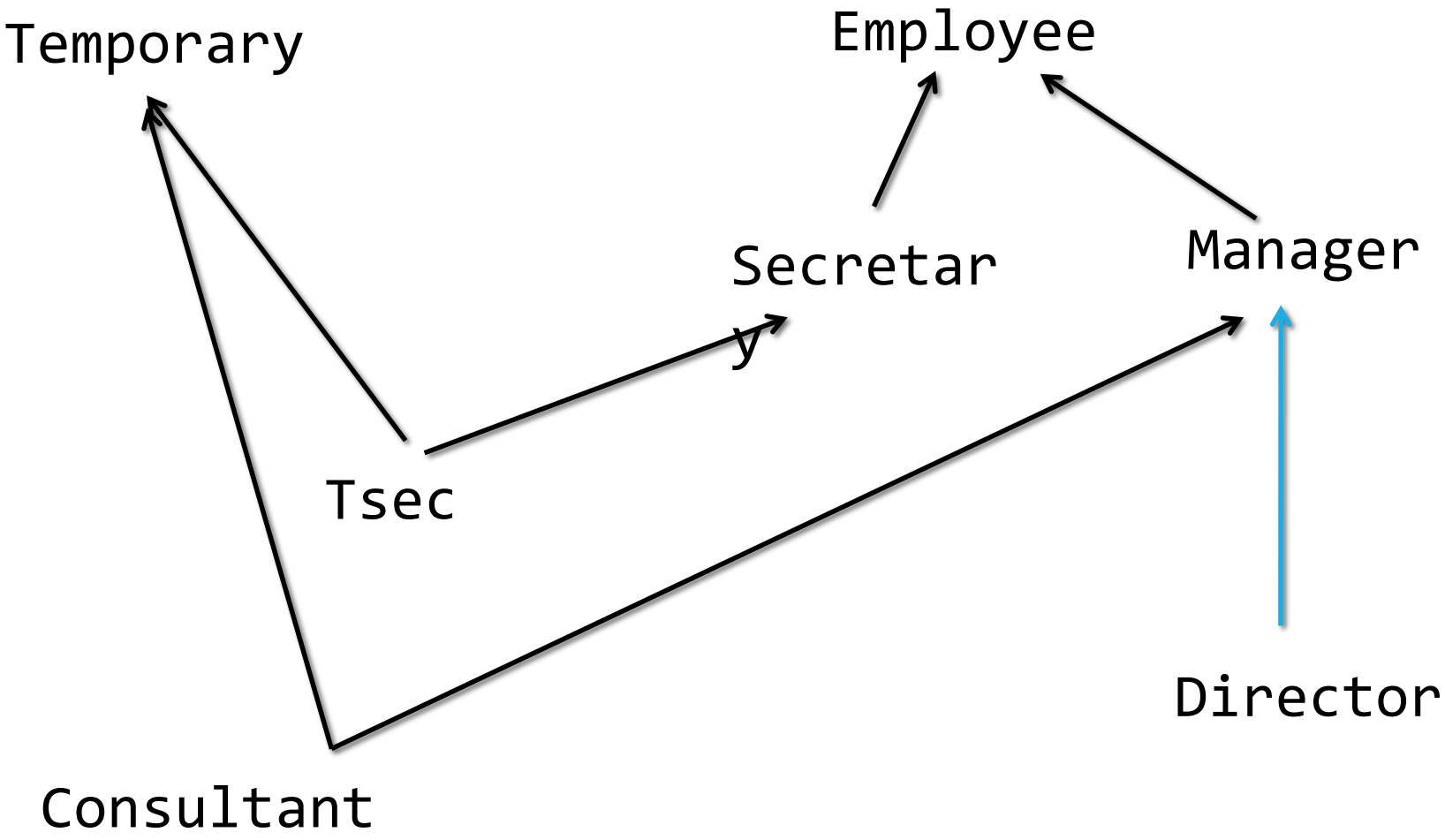
C++ példa (folyt.)

egy leszármazott lehet ős is

```
class Employee { … };
class Manager : public Employee {…};
class Director: public Manager {…};
```

Az osztályhierarchia lehet fa, de lehet általánosabb gráf is:

```
class Temporary { … };
class Secretary: public Employee { … };
class Tsec: public Temporary, public Secretary{ … };
class Consultant: public Temporary, public Manager { … };
```


Temporary
Employee
Secretar
y
Manager
Tsec
Director
Consultant

Implementáció újrahasznosítás: alosztályképzés

Használd egy típus implementációját egy másik típus implementálására!

Gyakran használjuk az őstípus implementációját az altípus implementálására

A gyakran használt OO programozási nyelvek keverik az altípus és alosztály fogalmakat:

- C++ - implementációs öröklés altípus nélkül:
  - `private`, `protected` öröklés

Mikor biztonságos az $S \subseteq T$ altípus reláció?

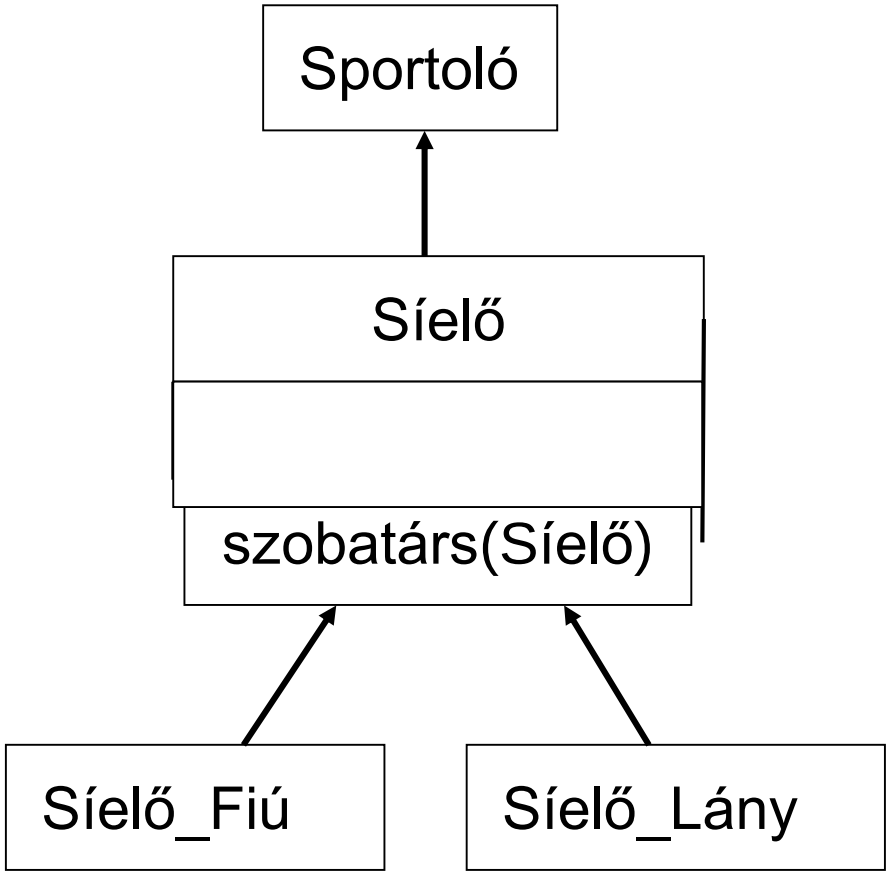
Az altípus metódusai megőrzik az őstípus viselkedését:

- Szignatúra:
  - Az argumentumok kontravarianciája, az eredmény kovarianciája
  - Az m_S kivételei benne vannak az m_T kivételei között
- Metódusok szabálya
  - Előfeltétel: “kontravariancia – altípus gyengébb”
  - Utófeltétel: “kovariancia – altípus erősebb”

Az altípusok megőrzik a szupertípusok tulajdonságait – típusinvariánsukra:

- “kovariancia – altípus erősebb”

Kontra/Ko-Variancia



Hogyan tudja Síelő_Lány felüldefiniálni szobatárs-at?

Kovariancia esetén:

szobatárs (Síelő_Lány) /

szobatárs (Síelő_Fiú)

Kontravariancia esetén:

szobatárs (Sportoló)

Novariancia esetén:

szobatárs (Síelő)

Probléma (kovariáns esetben):

s: Sielo; g: Sielo_Lany; b: Sielo_Fiu;

s := g; ... s. szobatárs (b);

Mit tesz a C++?

Lehet bevezetni kovariáns metódusokat az altípusban, de ezek túlterhelik az eredeti metódust, nem átdefiniálják!

- Példa:

```
class sielo {
   public:
      virtual void szobatars(sielo * s){
          cout<<"\n sielo szobatarsa sielo \n";
      };
};
```

```
class sielo_lany : public sielo {
  public:
     virtual void szobatars (sielo_lany *g){
         cout<<"\n sielo_lany szobatarsa lany ";
        };                          // túlterheli!
     virtual void szobatars (sielo *g){
          cout<<"\n szobatars sielo_lanyban” ;
      };    // átdefiniálja!
};
class sielo_fiu : public sielo {  … //hasonlóan }
```

```
void main(){
      sielo *s;
      sielo_lany *g;
      sielo_fiu *b;
      g= new sielo_lany;
      s = g;
      s->szobatars (b); //  szobatars sielo_lanyban
      g->szobatars (b); //  szobatars sielo_lanyban
      g->szobatars (g); // sielo_lany szobatarsa lany
      s->szobatars (g); // szobatars sielo_lanyban (!)
}
```

Többszörös öröklődés

Egy osztálynak egynél több közvetlen őse lehet

Problémák:

- Adattagok hányszor?
- Melyik metódus?

A többszörös öröklődés problémái:

```
class A{
int a;
public: virtual void f();
};
```

```
class B : public A {
 public: void f();};
```

```
class C : public A{
 public: void f();};
```

átdefiniál

átdefiniál

```
class D : public B, public C
{ …};
```

A többszörös öröklődés problémái:

1. Ha egy D-beli „f”-re (felüldefiniáltuk B-ben és/vagy C-ben) hivatkozunk, akkor az melyiket jelentse? (A v. B v. C)
2. Az „a” attribútum hány példányban jelenjen meg D-ben?

A két kérdés lényegében ugyanazt a problémát veti fel:
ha kétértelműség van, hogyan válasszunk?

Megoldási lehetőségek

A legtöbb esetben az ilyen kódot nem lehet lefordítani, a fordító, vagy a futtató környezet kétértelműségre (*ambiguous*) hivatkozva hibajelzéssel leáll.

Az ősosztály mondja meg, hogy mit szeretne tenni ilyen esetben.

A származtatott osztály mondja meg, hogy melyiket szeretné használni.

C++

A c++ „megoldása”

```
D d;
d.f();
```

- #error C2385: 'D::f' is ambiguous

vagy:

```
class D : public B, public C
{
public:
 using C::f;
};
```

vagy:

A a B és C virtuális bázisosztálya kell legyen ⇒

a –t (minden adattagot) csak egyszer örökli

C++ – többszörös öröklődés

```
class Animal {
 public: virtual void eat(); };

class Mammal : public Animal {
 public: virtual Color getHairColor();
 …};

class WingedAnimal : public Animal {
 public: virtual void flap();
 …};

// A bat is a winged mammal

class Bat : public Mammal, public WingedAnimal
 {
 …};

Bat bat;
```

Hogyan eszik??

C++ – többszörös öröklődés

```
class Mammal : public virtual Animal {
  public: virtual Color getHairColor();

 …};

class WingedAnimal : public virtual Animal {

    public: virtual void flap();
 …};

// A bat is still a winged mammal

class Bat : public Mammal, public WingedAnimal {
 …};
```

Absztrakt osztály

Tervezés eszköze

Egy felső szinten összefogja a közös tulajdonságokat

A metódusok között van olyan, aminek csak specifikációja van, törzse nincs

Nem hozható létre példánya.

A leszármazott teszi konkréttá.

Absztrakt osztály

```
class Alakzat{
public:
   virtual void kirajzol()=0;
   virtual bool zart()=0;
…
}
```

C++

```
class Sikidom {
    protected:
      int szelesseg, magassag;
    public:
     virtual int terulet()=0;
     void beallit_ertekek(int a, int b)
                           {szelesseg = a; magassag = b;}
};

class Teglalap: public Sikidom{
    public:
      int terulet()  {return (szelesseg * magassag);} …
};

class Haromszog: public Sikidom{
    public:
      int terulet()  {return (szelesseg * magassag /2);}
};
```

C++

Nem lehet:

```
Sikidom a;
```

Lehet:

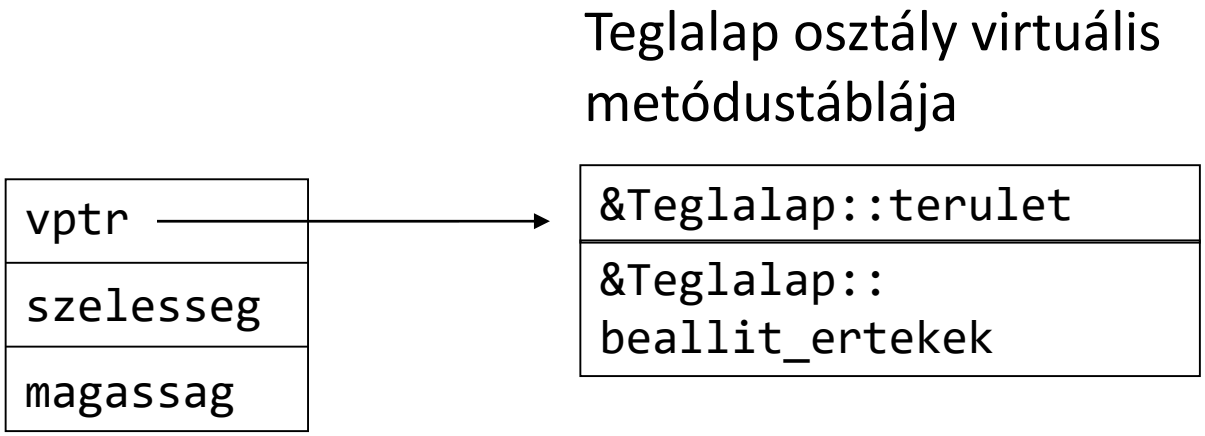
```
Teglalap t;
Haromszog h;

Sikidom* a2 = &t;
Sikidom* a3 = &h;

a2->terulet(); //Teglalap::terulet()
a3->terulet(); //Haromszog::terulet()
```

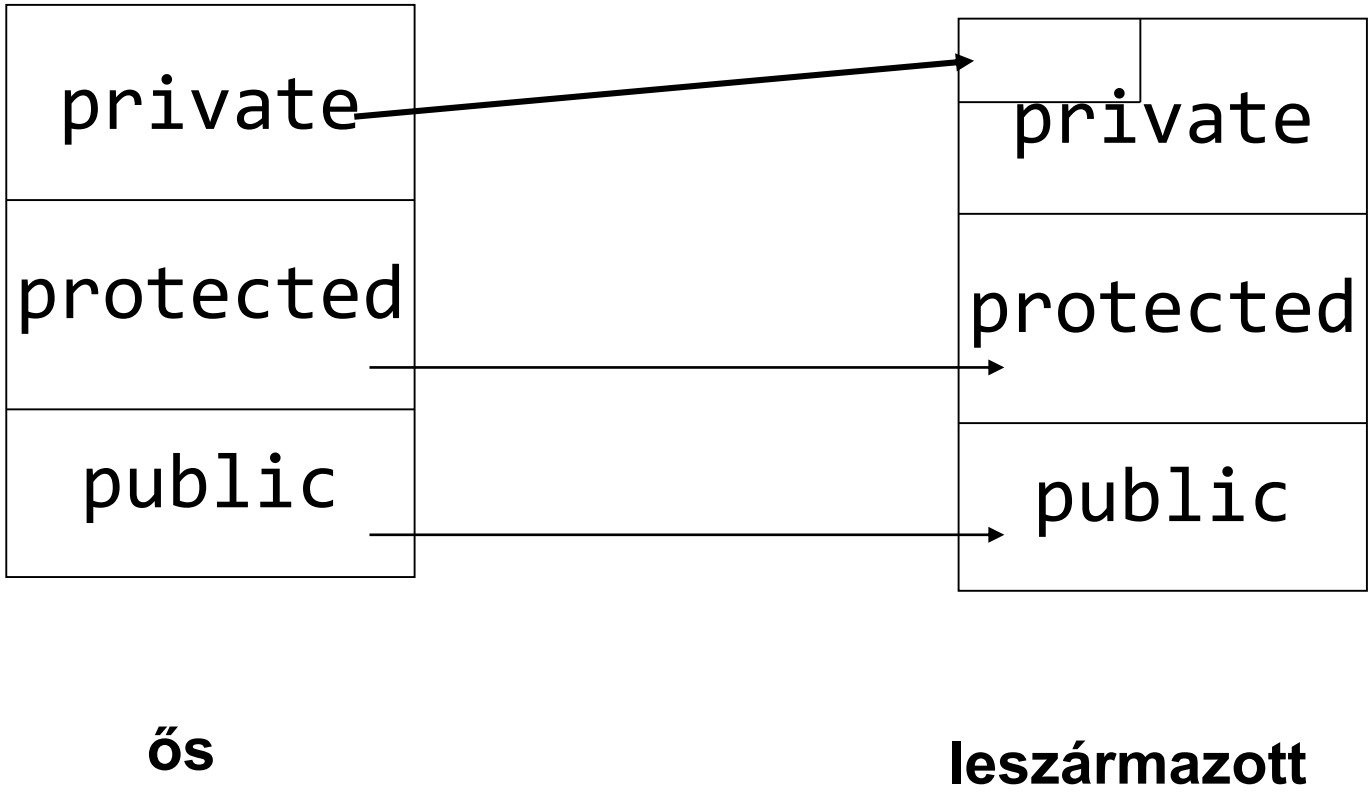
C++

Megvalósítás:
`Teglalap t;`



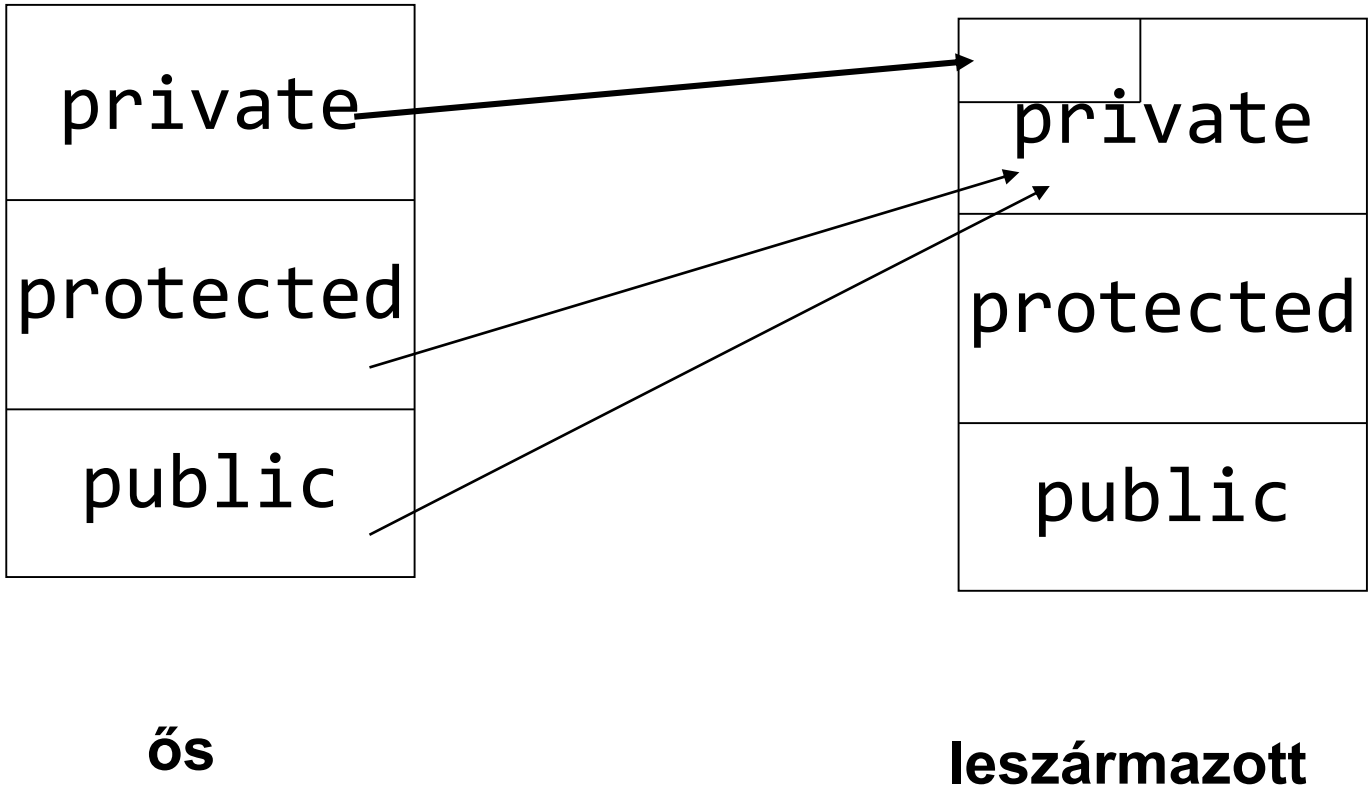
C++

public öröklődés:



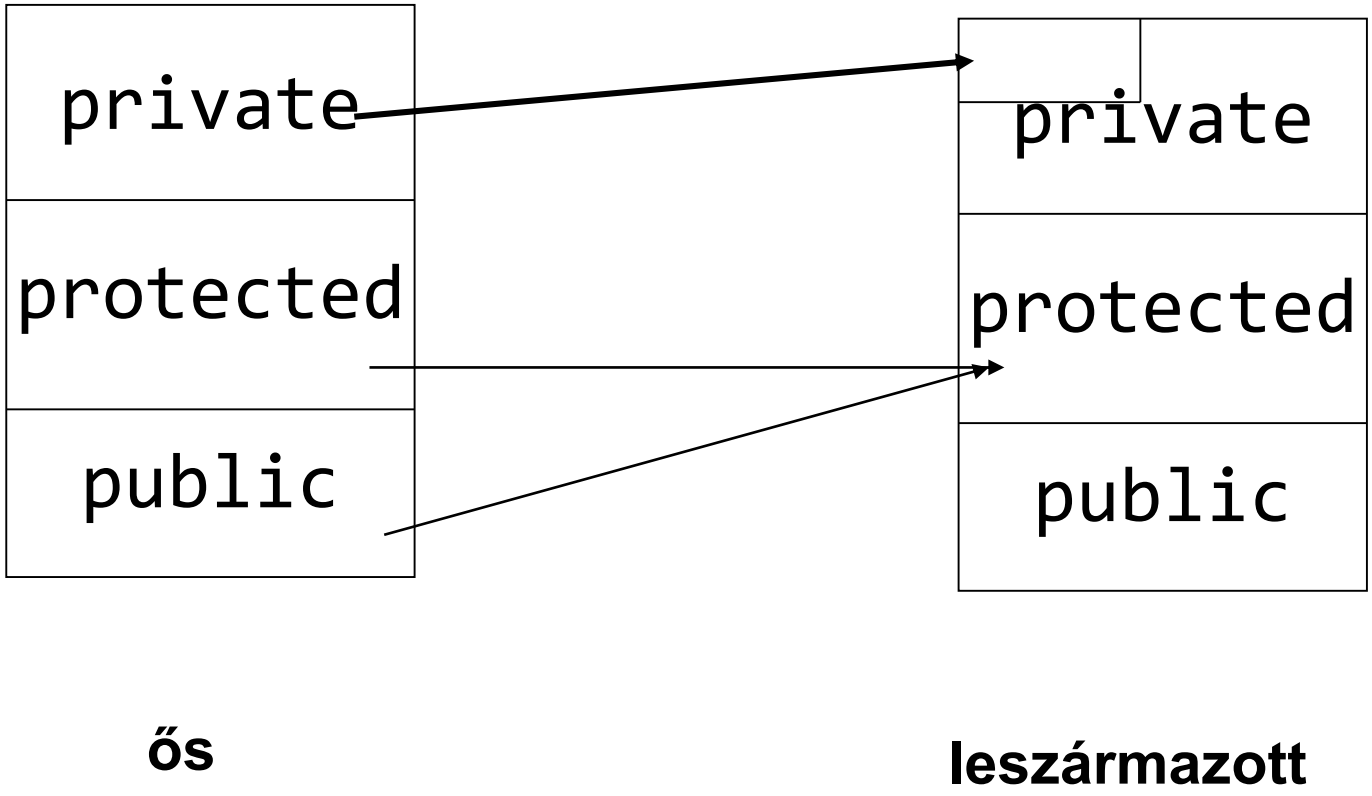
C++

private öröklődés:



C++

protected öröklődés:



C++

Mit örököl a leszármazott?

- Adattagokat
- Metódusokat

Mit nem örököl a leszármazott?

- Ősosztály konstruktorait, destruktorát
- Ősosztály értékadás operátorát
- Ősosztály barátait

C++

Mit vezethet be a leszármazott osztály?

- Új adattagokat
- Új metódusokat
- Felüldefiniálhat már meglévőket
- Új konstruktorokat és destruktort
- Új barátokat

C++

Egy általános metódus deklarációja a következőket jelenti:

1. A metódus elérheti a privát mezőket is
2. Az osztály scope-ját használja
3. A metódus egy konkrét objektumra hívódik meg, ezért birtokolja a ‘this’ pointert

Statikus metódus csak az 1, 2 - vel rendelkezik,

Ha egy függvényt friend-nek deklarálunk, akkor csak az 1. jogunk lesz (friend mechanizmus)

friend példa:

```
  typedef double Angle;
class Complex {
public:
 Complex(double r=0, double i=0){ R = r; I = i;}
 Complex operator =(Complex z){R = z.R; I = z.I; return *this; }
 Complex operator +(Complex z) {return  Complex(R+z.R,I+z.I);}
 Complex operator +(double x) { return Complex(R+x,I);}
 Complex operator *(Complex);
     Complex operator *(double);
     Complex operator -(Complex);
     Complex operator -(double);
     Complex operator /(Complex);
     Complex operator /(double);
     double Re(); double Im();
     double Abs(); Angle  Phi();
private:
 double R;  double I;
};
```

C++

```
Complex operator + (double x, Complex z){ return z+x;}
```

Vagy: osztály belsejében:

```
friend Complex operator+(double, Complex);
```

```
Complex operator+(double p1, Complex p2)
{
  Complex temp;
  temp.R = p1+p2.R;
  temp.I = p2.I;
  return (temp);
}
```

C++

Még egy (tipikus) friend példa

```
class Point {
  friend ostream &operator<<( ostream &, const Point &);
   public:
        Point( int = 0, int = 0 );       // default constructor
        void setPoint( int, int );       // set coordinates
        int getX() const { return x; }  // get x coordinate
        int getY() const { return y; }  // get y coordinate
    protected:          // accessible by derived classes
      int x, y;        // x and y coordinates of the Point
   }; // end class Point
```

Interfész

Típusspecifikáció támogatása

„Szolgáltatások”

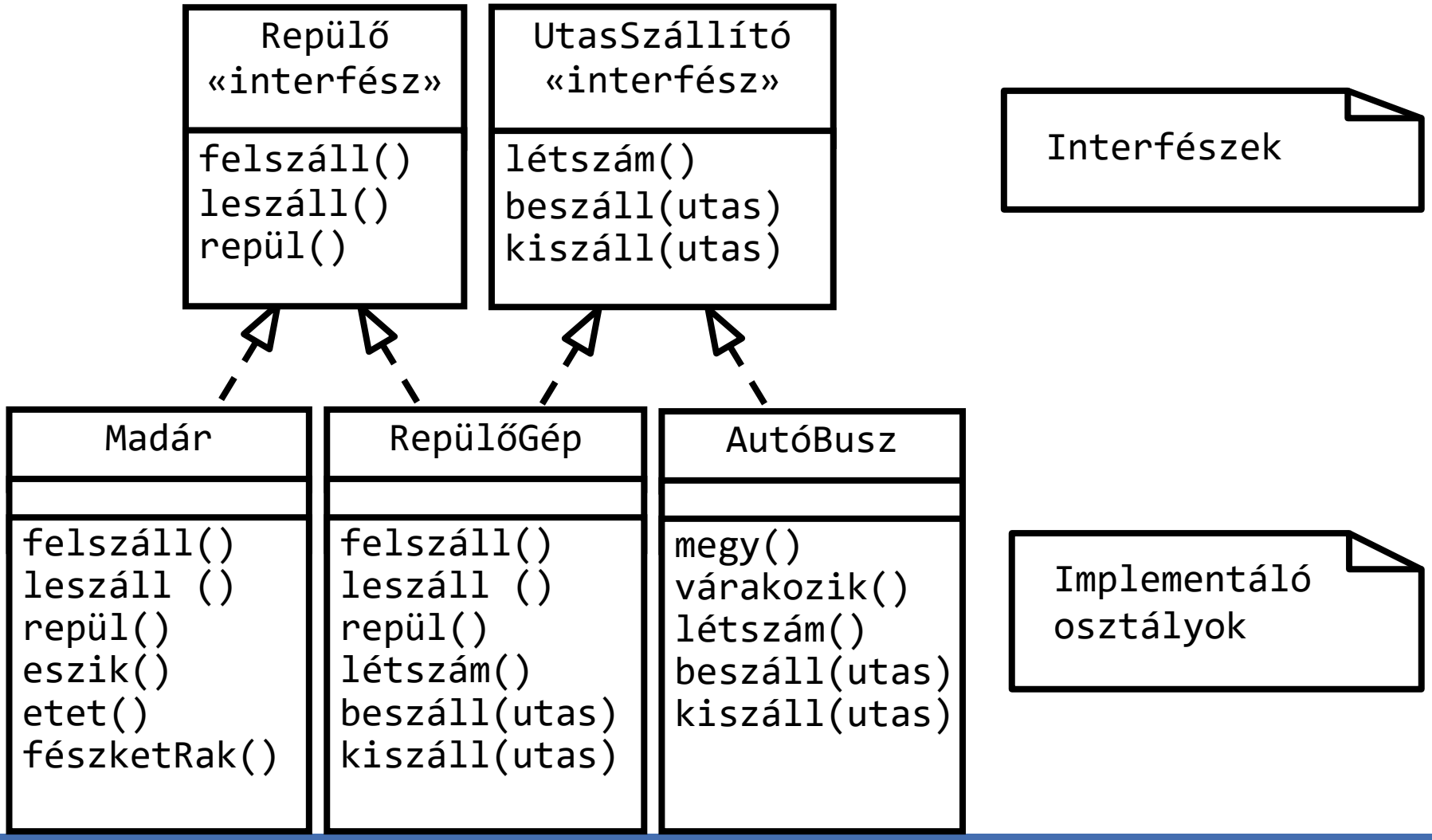
Meg kell valósítani a szolgáltatásait

Többszörös öröklődéssel nincs (annyi) probléma

- összefogja a közös jellemzőket

Interfész ≠ Absztrakt osztály!

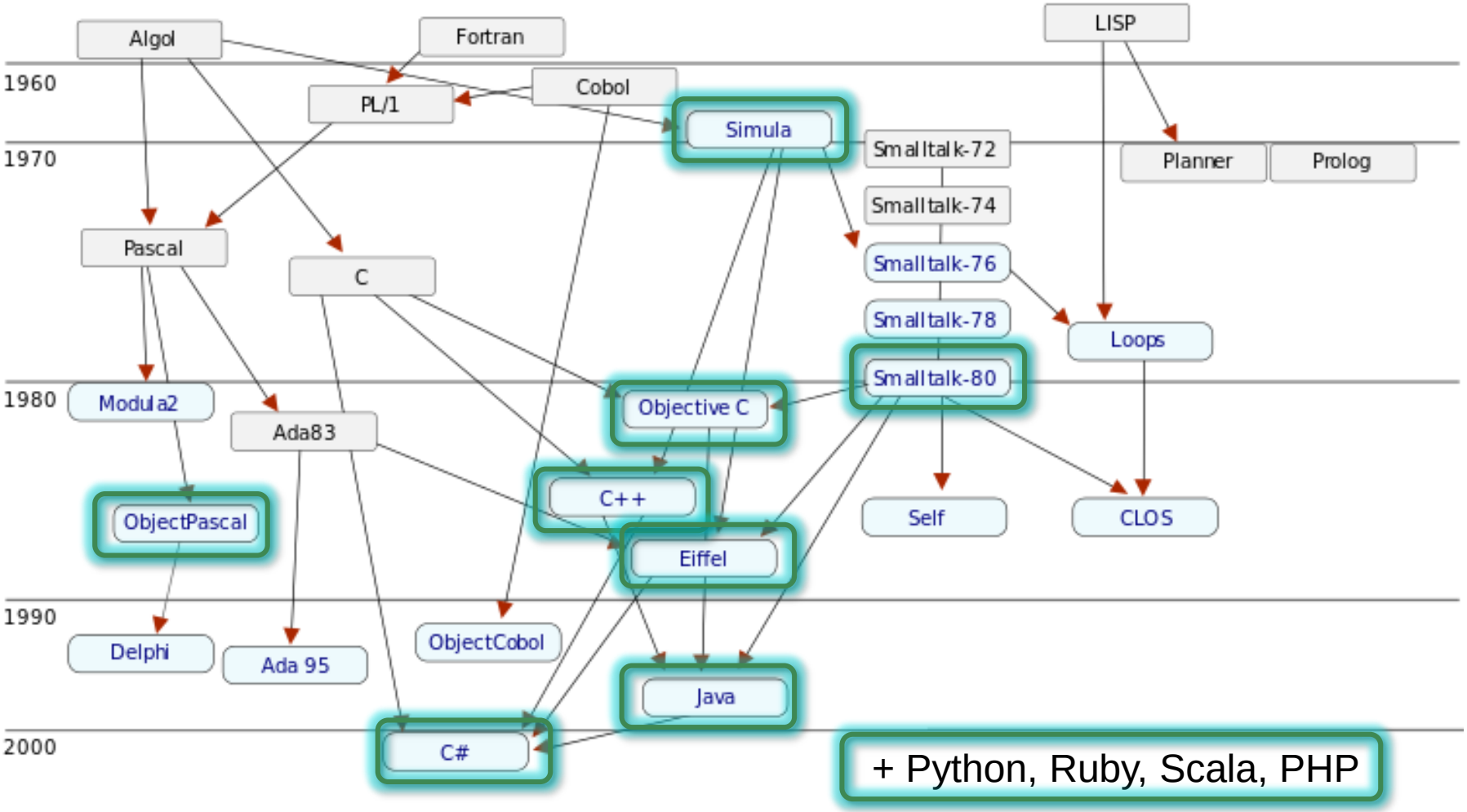
Interfész


Repülő
«interfész»
felszáll()
leszáll()
repül()
UtasSzállító
«interfész»
létszám()
beszáll(utas)
kiszáll(utas)
Interfészek
Madár
felszáll()
leszáll ()
repül()
eszik()
etet()
fészketRak()
RepülőGép
felszáll()
leszáll ()
repül()
létszám()
beszáll(utas)
kiszáll(utas)
AutóBusz
megy()
várakozik()
létszám()
beszáll(utas)
kiszáll(utas)
Implementáló
osztályok

További nyelvek

- OOP nyelvek jellemzői, szempontjai
- Programozási nyelvek OOP alapján
  - Simula 67
  - SmallTalk
  - C++
  - Object Pascal
  - Objective-C
  - Java
  - Eiffel
  - Python
  - Ruby
  - C#
  - Scala
  - PHP

Fejlődés, hatások



OOP fő elvei

OO moduláris struktúra

Adatabsztrakció

Osztályok

Öröklődés

Polimorfizmus és dinamikus összekapcsolás

Többszörös és ismételt öröklés

Automatikus memóriakezelés

Kérdések

Van-e öröklődés?

Van-e többszörös öröklődés?

Adattagok és metódusok elrejtése hogy van megoldva?

Támogatja-e a polimorfizmust és dinamikus összekapcsolást?

- Esetleg csak mutatók használatával?

Van-e alapértelmezett ősosztály?

- Object?

Kérdések

Van-e absztrakt osztály?

- Nem példányosítható közvetlenül?

Konstruktor

Destruktor?

- Garbage Collector?

Standard objektum könyvtárak?

Osztályszintű attribútumok és metódusok?

Simula 67

Az osztály (class) fogalom, valamint az öröklődés, az osztályhierarchia, először jelent meg itt

A blokkok prefixelése

- hatására a blokk és az osztály fogalma összeolvad
- A blokkban az osztályban definiált publikus fogalmakat használhatjuk (SIMULATION)

```
prefix_obj begin
...
end
```

Simula 67

```
class rendeles(szam);
  integer szam;
  begin
   integer egysegek_szama, erkezesi_idopont;
   real    feldolgozasi_ido;
  end;
```

A `rendeles` osztályhoz tartozó objektumot a `new rendeles(103)` kifejezéssel lehet létrehozni

Simula 67

öröklődés: „prefixeléssel”

```
rendeles class tetel_rendeles;
  begin
   integer tetel_meret;
   real feldolgozasi_ido;
   létrehozáskor lefutó utasítások
  end;
```

Simula 67

A teljesen önálló típusfogalom a SIMULA 67 nyelvben jelent meg először

- objektumhivatkozás: `ref (< osztályazonosító >)`

Bármely A osztály egyértelműen meghatároz egy `ref(A)` hivatkozási típust

- ez minden objektumhoz létezik.

Van GC

Nincs többszörös öröklődés

Van `this` pointer

További lehetőségek: `virtual`, `inner`

```
CLASS epulet(alapteruletX,alapteruletY); comment letrehozasi param;
  INTEGER alapteruletX,alapteruletY;     comment par-ek típusa;
  VIRTUAL :                              comment virt fv megadasa;
    PROCEDURE special_effects;
      ! Most jon az osztaly torzse;
      BEGIN
        PROCEDURE osszedol;
          BEGIN
            outtext("Bumm");
            special_effects;
            outtext("Bumm-bumm");
          END osszedol;
        PROCEDURE special_effects;
          BEGIN
            outtext("Fényeffektek");
          END special_effects;
    ! Letrehozasi utasitasok ide jonnek;
    …
    INNER;
    ! Az inner rész jelöli a leszarmazotthoz tartozo specialis;
    ! utasitasokat. Ha nem lenne akkor az itteni utasitasok után;
    ! hajtodnanak vegre.;
    …
    ! ide az jon amit meg ezutan csinalni kell.
END epulet; ! Az END es a pontosvesszo kozotti resz komment;
```

```
epulet CLASS garazs(autokszama);
  INTEGER autokszama;
  ! az epulet prefixkent irva a garazs ososztalya lesz;
  BEGIN
    PROCEDURE special_effects;

       ! Felulirja az ososztalybelit;
       BEGIN
         outtext("Autok szirenaznak");
       END special_effects;
       ! ide az jon ami az epulet osztaly-beli;
       ! 'inner' helyen fog vegrehajtodni a garazs;
       ! generalasakor ;
  END garazs;
```

Simula 67

Az inspect utasítás a dinamikus összekapcsolás megvalósítása mellett:

- `X in osztály_1`

```
inspect X

when osztály_1 do utasítás_1

when osztály_2 do utasítás_2 ...

when osztály_n do utasítás_n otherwise utasítás_0
```

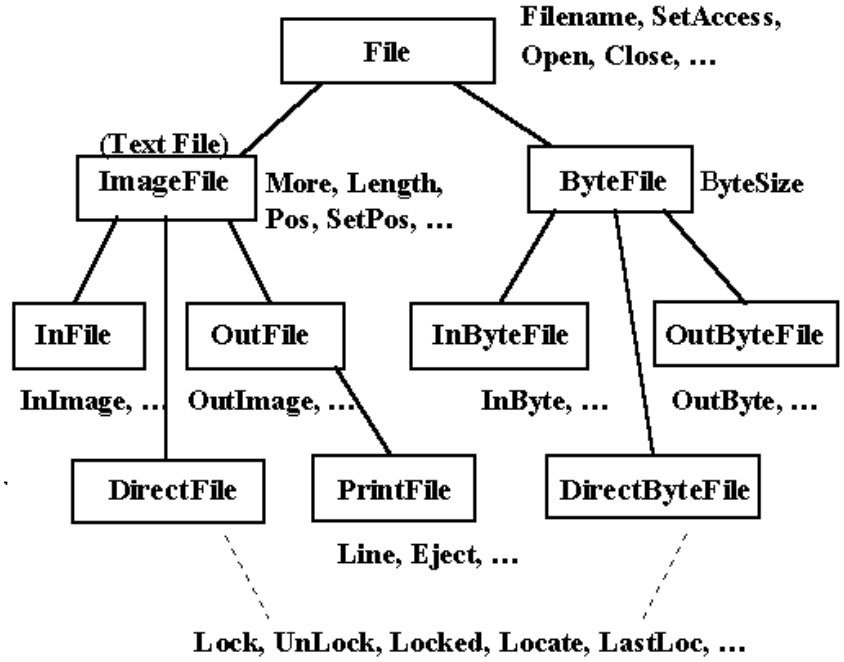
Simula 67

Láthatósági védelem

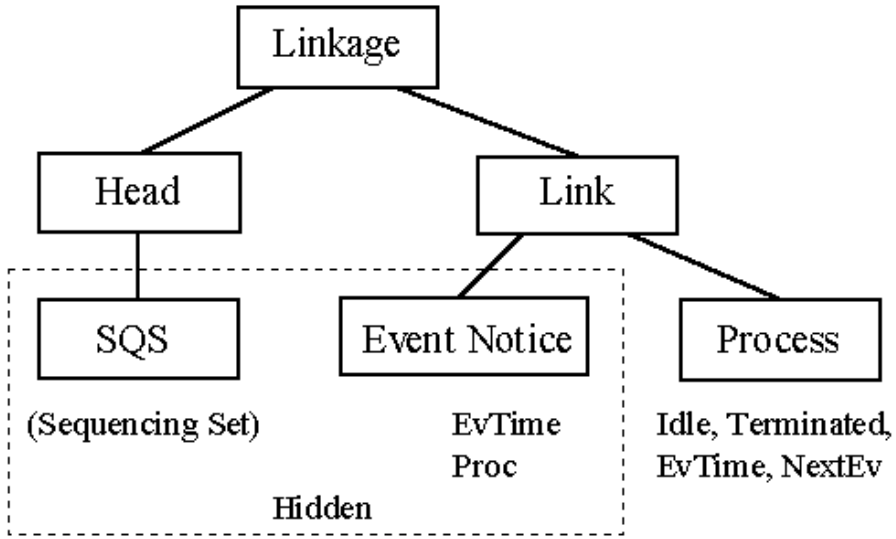
- hidden és protected prefixek az attribútumok előtt

Standard könyvtárak

- BASICIO, FILE



- SIMULATION, PROCESS



SmallTalk

Minden objektum

- Integer,…Character, még az üzenetek - amelyeket az objektumoknak küldünk - is objektumok
- Még az IDE is: ClassBrowser, Workspace, Debugger)

Egy változóról nem tudjuk, hogy milyen típusú objektumra mutat, a változók típusnélküliek

- RTTI (run-time type information)

Szabványos objektumkönyvtárak

- (Array, String, File Stream, stb.)

VM

GC

SmallTalk

A ST szintaxisa 1 nap alatt megtanulható

- ennek 90%-át 1 perc alatt meg lehet érteni, ez pedig a következő:
  - `object message`
  - Példák
    - `myWindow drawCircle,`
    - `myAge + 1,`
    - `myWindow drawCircleofRadius: afloat, color: Red`

SmallTalk

Legnagyobb közös osztó keresése:

```
| a b |
a := 345.
b := 230.
[ a ~= b ] whileTrue:
  [ a < b ifTrue:
           [ b := b - a ]
         ifFalse:
           [ a := a - b ]
  ].
^a
```

Magyarázat

ciklus: logikai értéket visszaadó blokk objektumnak küldjük

- a `whileTrue:` üzenetet, aminek az argumentuma is egy blokk

elágazás: logikai értéknek küldjük

- az `ifTrue: ifFalse:` üzenetet, blokkok az argumentumok

SmallTalk

Minden objektum

- Minden objektum egy osztály példánya

Az osztály is objektum

- Kérdés: ez minek a példánya?
- Válasz: egy metaosztály példánya
  - (metaosztály hierarchia)

SmallTalk

Az objektumok osztályát a class üzenettel kapjuk vissza

Osztályok definiálása a szülőnek küldött üzenettel történik (subclass)

Nincs többszörös öröklődés

- nincs anomália

Láthatóság

- nincsenek láthatósági predikátumok
- a belső változók minden alosztály számára láthatóak
- De kívülre nem, viszont a metódusok globálisak
  - konvenció: a megjegyzésbe private-t írhatunk

Van osztályszintű attribútum és metódus

SmallTalk

Közös ősosztály
- Object
  - objektum kiíratása
  - objektum osztályának meghatározása
  - copy
  - klónozás

Konstruktor
- osztálynak küldött üzenet

Destruktor nincs
- szemétgyűjtéssel oldja meg

Nincs absztrakt osztály, azaz a metódusokat definiálni kell, nem csak deklarálni

SmallTalk

```
Object subclass #Szamla
  instanceVariableNames: 'egyenleg'
  classVariableNames: ''
  poolDictionaries: ''
  category: nil !.

!Szamla class methodsFor: 'instance creation'!
  new
    |r|
    r := super new.
      r init.
      ^r

!Szamla methodsFor:       'instance initialization'!
  Init
   egyenleg := 0
  !!
```

C++

Az elméleti bevezető során szerepelt.

Object Pascal

Többféle következetlenség

- `object – class`
- `virtual` (sebesség) - `dynamic` (méret)
- Láthatósági megkötések alkalmazhatóak – de fájlon belül nem érvényesek!
  - De: strict private, strict protected lehetőség is van

Új láthatóság: published (publikált)

- ugyanazon elérési szabályok, mint a public részekre, de a fordító futási idejű típus-információkat is kapcsol hozzá

Jó ötlet: property

- `property p: word read olvas write ír default 5;`

class kulcsszó

Automatikus memóriakezelés

- A Delphi referencia modellt használ az osztályok esetén, az ilyen objektum mindig dinamikus, implicit referencia.

Közös ős

- Minden osztály a `TObject` ősosztály leszármazottja.

Absztrakt metódusok

- Ha a metódus deklarációjában az `abstract` kulcsszót használjuk, nem kell törzset megadni – de az ilyen osztálynak is lehet példánya!

Object Pascal

```
type
 TAnimal = class
  public
   constructor Create;
   function Szovege: string; virtual;  // virtuális metódus
   function SzovegeStatic: string;     // statikus metódus
  end;

 TDog = class (TAnimal)
  public
   constructor Create;
   function Szovege: string; override;
       // felüldefiniáljuk a TAnimal virtuális metódusát
   function SzovegeStatic: string;
       // felüldefiniáljuk a TAnimal statikus metódusát
  end;
```

Object Pascal

A függvények megvalósítása:

```
function TAnimal.Szovege : string;
  //a direktívákat a kifejtésnél már nem szabad megadni
begin
     Result:='nem tudom';
end;

function TAnimal.SzovegeStatic : string;
begin
     Result:='nem tudom nem tudom';
end;

function TDog.Szovege : string;
begin
     Result:='vau vau';
end;

function TDog.SzovegeStatic : string;
begin
     Result:='vau';
end;
```

Object Pascal

```
var
 Animal1, Animal2: TAnimal;
 Dog1: TDog;
begin
 Animal1:=TAnimal.Create;
 Animal2:=TDog.Create;
 Dog1:=TDog.Create;
 ShowMessage( Animal1.Szovege);
  // a megjelenített ablak szövege: 'nem tudom'
 ShowMessage(Animal1.SzovegeStatic);
  // a megjelenített ablak szövege: 'nem tudom nem tudom'
 ShowMessage( Animal2.Szovege);
  // a megjelenített ablak szövege: 'vau vau'
```

Object Pascal

```
ShowMessage(Animal2.SzovegeStatic);
    // a megjelenített ablak szövege: 'nem tudom nem tudom'
    // oka: a metódus statikus volt, és mi most egy TAnimal
    // statikus típusú változó statikus metódusát hívtuk

ShowMessage( Dog1.SzovegeStatic);
    // a megjelenített ablak szövege: 'vau vau vau vau'
    // oka: most a TDog statikus metódusát hívtuk
Animal1:=Dog1;  // ez a polimorfizmus miatt ok.

ShowMessage(Animal1.SzovegeStatic);
    // a megjelenített ablak szöveg: 'nem tudom nem tudom'
    // oka: a statikus metódust egy TAnimal statikus típusú
    // változóra hívtuk meg
end;
```

Object Pascal

```
type TOs = class
   a, b: Integer;
   constructor Letrehoz(x, y: Integer);
   ...
end;

type TUj = class(Tos)
   c: Integer;
   constructor Letrehoz(x, y, z: Integer);
   ...
end;
```

Object Pascal

```
constructor TUj.Letrehoz(x, y, z: Integer);
   begin
     inherited Letrehoz(x, y);
      // itt a TOs Letrehoz konstruktorát hívjuk
     c:=z;
   end;
```

Object Pascal – absztrakt osztály

```
type
 TPolygon = class
 private
   sideLength : Byte;
   sideCount  : Byte;
   function  GetSideLength : Byte;
   procedure SetSideLength(length : Byte);
   function  GetSideCount : Byte;         virtual; abstract;
   procedure SetSideCount(count : Byte); virtual; abstract;
   function  GetArea : Single;            virtual; abstract;
  published
   property length : Byte
       read GetSideLength         write SetSideLength;
   property count : Byte
       read GetSideCount      write SetSideCount;
   constructor Create(length : Byte);  virtual;    //!!
  end;
```

Object Pascal – absztrakt osztály

```
function TPolygon.GetSideLength : Byte;
begin
  Result := sideLength;

end;

procedure TPolygon.SetSideLength(length : Byte);
begin
  sideLength := length;
end;

constructor TPolygon.Create(length : Byte);
begin
  sideLength := length;
end;
```

Object Pascal – absztrakt osztály

```
type
  TTriangle = class(TPolygon)
  private
    function  GetSideCount : Byte;         override;
    procedure SetSideCount(count : Byte); override;
    function  GetArea : Single;            override;
  published
    constructor Create(length : Byte);     override;
  end;
```

Object Pascal – absztrakt osztály

```
function TTriangle.GetSideCount : Byte;
begin
  Result := sideCount;
end;

procedure TTriangle.SetSideCount(count : Byte);
begin
  sideCount := count;

end;

function TTriangle.GetArea : Single;
begin
  Result := sideLength * sideLength * SQRT(3)/4;
end;

constructor TTriangle.Create(length : Byte);
begin
  sideLength := length;
  sideCount := 3;
end;
```

Object Pascal – absztrakt osztály

```
TSquare = class(TPolygon)
  private
    function  GetSideCount : Byte;          override;
    procedure SetSideCount(count : Byte); override;
    function  GetArea : Single;             override;
  published
    constructor Create(length : Byte);     override;
  end;
```

Object Pascal – virtuális konstruktor

```
TFoo = class
 public
  constructor Create; virtual;
 end;


TBar = class(TFoo);
 public
  constructor Create; override;
 end;
```

Object Pascal – virtuális konstruktor

```
TFooClass = class of TFoo; //declares a class reference type
 ...
 var
  fc: TFooClass;
  afoo: TFoo;

 begin
  fc := TFoo;
  afoo := fc.Create; // returns a TFoo
 ...
  fc := TBar;
  afoo := fc.Create; // returns a TBar
```

Object Pascal – sealed osztály

sealed – nem származtatható belőle

```
type
  TMyClass = class sealed
    procedure SomeProcedure; …
end;
```

final – ha metódus

```
procedure MyMethod(const AMyParameter: Integer);
override; final;
```

Object Pascal – interfészek

```
type
IMozgathato = interface
      procedure Mozgat(x,y: integer);
end;
```

Az interface-ben:

- csak a metódusok fejrésze van leírva
- tartalmazhat property-ket is, de ezek szintén nincsenek kidolgozva, csak a property neve van megadva, típusa, és az, hogy írható vagy olvasható,
- mezőket, konstruktort, destruktort nem tartalmazhat
- minden rész az interface-ben automatikusan publikus,
- a metódusokat nem lehet megjelölni virtual, dynamic, abstract, override kulcsszavakkal,
- az interface-eknek lehetnek ősei, melyek szintén interface-ek.

Object Pascal – interfészek

```
Type TKor = class(IMozgathato)
  public
     procedure Mozgat(x,y: integer);
  end;

TLabda = class(IMozgathato)
  public
     procedure Mozgat(x,y: integer);
  end;

procedure ArrebRak(x: IMozgathato)
begin
     x.Mozgat(12,17);
end;
```

Objective-C

Objektumorientált nyelv

- a C programozási nyelvet egészíti ki
- a Smalltalk üzenetküldési lehetőségeivel.

1986-ban jelent meg, Brad Cox és Tom Love tervezték

Apple által továbbfejlesztett és támogatott

- Mostanra megvan az utódja is …

Objective-C

Osztályok

- Kötelező szétválasztani az interfészt és az implementációt.
- Az interfész deklarálja az osztály adattagjait, metódusait, és megnevezi az ősosztályát, vagyis ez a specifikáció!
- Az implementációban pedig definiáljuk a metódusokat, ezzel tulajdonképpen az osztályt

Objective-C

Az osztály interfész része

```
@interface ClassName : ItsSuperclass
{
        float width;
        float height;
        BOOL filled;
        NSColor *fillColor;
        //...
}   //itt az adattagok
        + alloc;    // osztálymetódus előtt +
        - (void)display; // példánymetódus előtt –
        - (void)setWidth:(float)width height:(float)height;
        - makeGroup:group, ...;
@end
```

Objective-C

Az osztály implementáció része

```
#import "ClassName.h„
@implementation ClassName
   + alloc
   {               //...
   }
   - (void)display
   {               //...
   }
   - (void)setWidth:(float)width height:(float)height
   {               //...
   }
   - makeGroup:group, ...
   {
         va_list ap;
         va_start(ap, group);
         //...
   }
@end
```

Üzenetküldés

Itt a metódusok/függvények hívása helyett az objektumok közötti üzenetküldés kerül előtérbe

- `[foo bar:parameter];`

Az hogy az üzenetet fel tudja-e dolgozni az objektum, az futásidőben dől el

- Ahogyan a típusellenőrzés sem történik meg, csak futásidőben

Az üzenetküldés során a paraméterek a függvény nevével adhatók meg

```
-(type)method:(type)param1 andParam2:(típus)param2;
```

Objective-C

Láthatóság szabályozása a következő direktívákkal lehetséges

- `@public`
- `@private`
- `@protected`
  - Alapértelmezett
- `@package`

Objective-C

Öröklődés

- Egyszeres öröklődés
- `NSObject` a legfelső szinten

Interface

- @protocol

polimorfizmus, dinamikus kötés – automatikusan

Objective-C

```
@protocol Archiving
- read: (FILE *) f;
- write: (FILE *) f;
@end

@protocol ReferenceCounting
- incrementCount;
- decrementCount;
- (unsigned) refCount;
@end
//.....................................
@interface Shape : NSObject
<Archiving, ReferenceCounting>
...
```

Objective-C

Kategóriák

- A kategória segítségével már meglévő osztályokhoz adhatunk hozzá metódusokat
  - akkor is, ha nincs meg az osztály forrása
- Ez egy rendkívül erős eszköz, amellyel öröklés nélkül terjeszthetjük ki az osztályok funkcionalitását
  - akár beépített osztályokét is

Objective-C

```
#import "ClassName.h"
@interface ClassName ( CategoryName )
// method declarations
@end
 .....................................
#import "ClassName+CategoryName.h"  // ilyen neve legyen!
@implementation ClassName ( CategoryName )
// method definitions
@end
```

Objective-C

Példa – NSString minden objektumának legyen isURL metódusa:

```
@interface NSString (Utilities)
- (BOOL) isURL;
@end
```

Egy implementációja lehet:

```
#import "NSString+Utilities.h"
@implementation NSString (Utilities)
- (BOOL) isURL
{
    if ( [self hasPrefix:@"http://"] )
            return YES;
    else
            return NO;
}
@end
```

Objective-C

Most már bármelyik NSString-re alkalmazhatjuk ezt a metódust

```
NSString* string1 = @"http://pixar.com/";
NSString* string2 = @"Pixar";
if ( [string1 isURL] )
NSLog (@"a string1 egy URL");
if ( [string2 isURL] )
NSLog (@"a string2 egy URL");
- állapotváltozó nem adható így hozzá, csak
öröklődéssel!
```

Java

Smalltalk-kal rokonság

Van garbage collector (VM dolga)

Objektumelvűség

- A Javában gyakorlatilag minden objektum(osztály)

Nincs osztályon kívüli (globális) változó

Öröklődési fa gyökere az Object osztály

Nincs többszörös öröklődés

Java

```
public class Alkalmazott{
	String nev;
	int fizetes;
	…
	public void fizetestEmel(int novekmeny){
		fizetes += novekmeny;
	}
}

Alkalmazott a;
a = new Alkalmazott();
…
a.fizetestEmel(60000);
```

Java

```
public class Fonok extends Alkalmazott{
    final int MAXBEOSZT = 20;
    Alkalmazott[] beosztottak =
                new Alkalmazott[MAXBEOSZT];
    int beosztottakSzama = 0;

    public void ujBeosztott(Alkalmazott b) {
     …
    }
}
```

Java

Többszörös öröklődés helyett van interface

- `extends` - `implements`

Szabványos osztálykönyvtárak!

Beépített típusoknak (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`) csomagoló osztálya (“wrappere”) van, amely felelős a konverzióért, kiíratásért, stb. – ezek immutable osztályok

- `boolean` -> `Boolean` (nagybetű a különbség)
- `char` ->`Character`
- `int` -> `Integer`

`instanceof`

- új operátor a C++ -hoz képest

Java

final, mint módosító

Osztálynév előtt: tiltja a további származtatást

Metódus előtt: tiltja a felüldefiniálást

Változó előtt: tiltja a kezdeti értékadás után az érték megváltoztatását

- Objektum esetén referencia

Java

Abstract, mint módosító

Metódus előtt: a metódus deklarálható, de nem definiálható az adott osztályban, csak a származtatottban

Osztály előtt: az osztálynak van abstract metódusa (nem kötelező kiírni, de az osztály akkor is abstract lesz implicite)

Egy osztály nem lehet egyszerre final és abstract

Java

static, mint módosító

```
public class osztv {
    public osztv(){
        pldszam++;
    }
    static int pldszam=0;
    public static int get_pldszam() {
        return pldszam;
    }
}
```

Java

static, mint módosító

```
public static void main(String[] args) {
      osztv o1 =new osztv();
       System.out.println(o1.get_pldszam() + " az objszám " );
      megegy();
      System.out.println(osztv.get_pldszam() + " az objszám most " );
}

protected static osztv megegy(){
      osztv o2 = new osztv();
      return o2;
}
```

Java

nincs operátor túlterhelés (de túlterhelés van)

Ha egy osztálynak nincs konstruktora, a fordító a következőt generálja neki:

- ```
  class ÉnOsztályom extends MásikOsztály {
       ÉnOsztályom() {  super();  }
  }
  ```

Destruktorok: a nyelvben nem szerepelnek

- helyette a finalize() (a GC hívja felszámolás előtt)

Interface: metódusok egy csoportját specifikálja, törzsük implementálása nélkül

- konstans változókat deklarálhatunk benne
- többszörös interface öröklés engedélyezett
  ```

Java

Anomáliák: egyszerűbb a helyzet az interfészek miatt

- „melyik f() implementáció fusson le?”
  (csak egy implementáció lehet)

Interfészek:

```
interface Savanyu{ }
public interface Gyumolcs{
    int IZ = 1;
    void egyedMeg();
}
interface Alma extends Gyumolcs{
    int SZIN = 2;
    int milyenSzinu();
}
```

Java

Interfészek öröklődése:

```
interface Narancs extends Gyumolcs, Savanyu{
    int MERET=1;
}
```

Implementálása:

```
class Golden implements Alma{
    public void egyedMeg() {/*...*/}
    public int milyenSzinu() {/*...*/}
}
```

Java – default interface

```
public interface Addressable
{
    String getStreet();
    String getCity();

    default String getFullAddress()
    {
        return getStreet()+", "+getCity();
    }
}
```

Java – default interface

```
public class Letter implements Addressable
{
   private String street;
   private String city;
   public Letter(String street, String city)
   {
      this.street = street;
      this.city = city;
   }
   @Override
   public String getCity()
   {
      return city;
   }
   @Override
   public String getStreet()
   {
      return street;
   }
   public static void main(String[] args)
   {
      Letter l = new Letter("123 AnyStreet", "AnyCity");
      System.out.println(l.getFullAddress());
   }
}
```

Java – default interface – többszörös implementáció

```
public class TheClass
implements Interface1,
Interface2 {


    public void doIt()
    {
        Interface2.super.doIt();
        Interface1.super.doIt();
    }
}
```

Felül kell definiálni!

Valamint explicit módon kell jelezni, hogy melyik ős-interfész megoldását hívjuk meg.

Java – default interface – többszörös implementáció

```
public interface Interface2 {

    default void doIt(){
        System.out.println("I2.doIt");
    }
}
public interface Interface1 {

    default void doIt(){
        System.out.println("I1.doIt");
    }
}
```