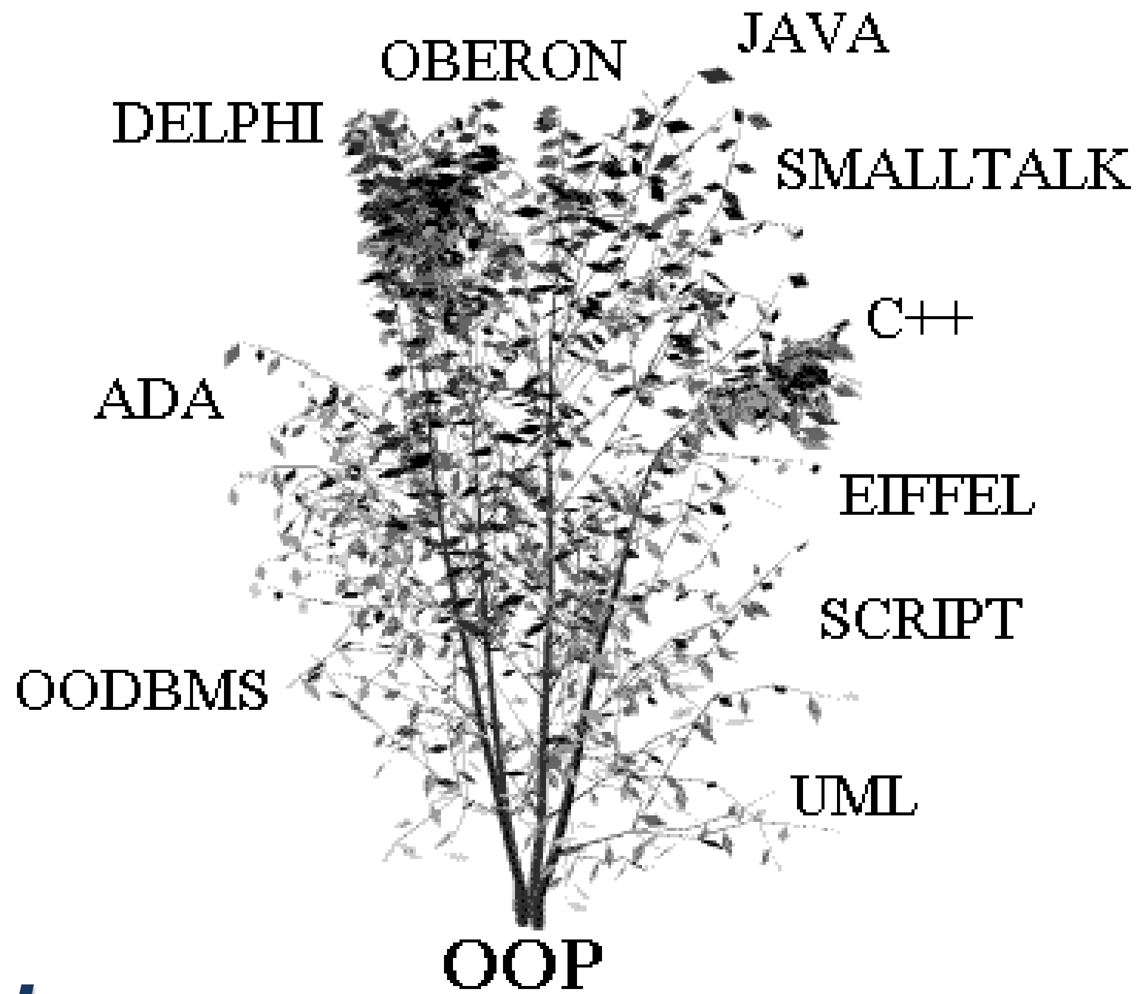
Objektum-orientált programozás 2. rész



Nyelvek

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: pl. Object?

- Van-e absztrakt osztály? (Ugye nem példányosítható közvetlenül?)
- konstruktor – destruktor?
- GC?
- 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
- 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("Nyiiii-kreccs-bumm");
               special_effects;
               outtext("DRBRBFPAABAAAAAAAAAAAAAANG!");
             END osszedol;
           PROCEDURE special_effects;
             BEGIN
               outtext("Menekülési es halálozási hangok hallatszanak.");
             END special_effects;
        ... ide az jon amit letrehozaskor csinalni kell.
        INNER; ! itt hajtodnak vegre a leszarmazotthoz tartozo specialis
                  dolgok.Ha nem lenne akkor az itteni utasitasok utan
    hajtodnanak vegre.;
        ... ide az jon amit meg ezutan csinalni kell.
END epulet; !Az END es a pontosvesszo kozotti resz komment;
```

öröklődés:

```
epulet CLASS tyukol(tyukszam, kakasszam);
    INTEGER tyukszam,kakasszam;
   ! az epulet prefixkent irva a tyukol ososztalya lesz;
      BEGIN
        PROCEDURE special_effects;
                         !  Felulirja az ososztalybelit;
          BEGIN
            outtext("Kotkothajajajjjj-kukurikuuuuu-
                 aaaaaaaaajajjjjjj");
          END special_effects;
          ...
         ide az jon ami az epulet osztaly-beli
           'inner' helyen fog vegrehajtodni
          tyukol generalasakor
    END tyukol;
```

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:
  pl. BASICIO, FILE,
  SIMULATION, PROCESS
- Szimulációs célok használatát a nyelvek.inf.elte.hu portálon lévő példaprogramok mutatják

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

Point>><= (comparing, public, <magnitude>)

File Edit Workspace Class Method History Tools Window Help

Link
ListViewColumn
Locale
Magnitude
ArithmeticValue
123 Number
3.14 Float
3/4 Fraction
123 Integer
5.99 ScaledDecimal
Point
Point3D
Association

Instance | Class

All
*
accessing
arithmetic
coercing
comparing
converting
double dispatch
mathematical
operations

Categories | Protocols | Variables

Methods
<
<=
=
>
>=
hash
max:
min:

Method source | Class definition | Class comment | Class Diagram

```
<= anArithmeticValue
	"Answer whether the receiver is neither below nor to the right of anArithmeticValue.
	A double dispatch of #< with the superclass implementation of #<= would not work here."

	| aPoint |
	aPoint := anArithmeticValue asPoint.
	^x <= aPoint x and: [ y <= aPoint y ]
```

Dolphin

SmallTalk

- a ST szintaxisa 1 nap alatt megtanulható
- ennek 90%-át 1 perc alatt meg lehet érteni, ez pedig a következő:

így kódolunk ST-ben:

object message

myWindow drawCircle,

myAge + 1,

myWindow drawCircleofRadius: afloat, <u>color:</u> 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
```

- **ciklus**: logikai értéket visszaadó blokk objektumnak küldjük pl. a whileTrue: üzenetet, aminek az argumentuma is egy blokk

- **elágazás**: logikai értéknek küldjük pl. 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 (obj. kiíratása, obj. 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++

- C nyelv egyik OOP kiterjesztése
- Simula-féle objektumelvűséget követi (majdnem…)
- nincs garbage collector
- az adattagok és metódusok elrejtése megoldott
- a láthatóság minősítője lehet:
  - public
  - protected
  - private

C++

- public - a külső felhasználók elérik
- protected – csak a leszármazottak érhetik el
- private – csak az adott osztály és „barátai” számára elérhető – alapértelmezés az osztályoknál

C++

```
class Teglalap{
    int x, y;
  public:
    void ertekadas(int, int);
    int terulet() {return (x*y);}
};
```

C++

```
void Teglalap::ertekadas(int x1, int y1){
  x = x1;
  y = y1;
 }
Teglalap haz; //statikus helyfoglalású példány!
 haz.ertekadas(5,3);
 int thaz = haz.terulet(); // thaz = 15

 Teglalap* kert = new Teglalap;
                    //dinamikus helyfoglalású példány
 kert->ertekadas(20,17);
 int tkert = kert->terulet(); //tkert = 340
```

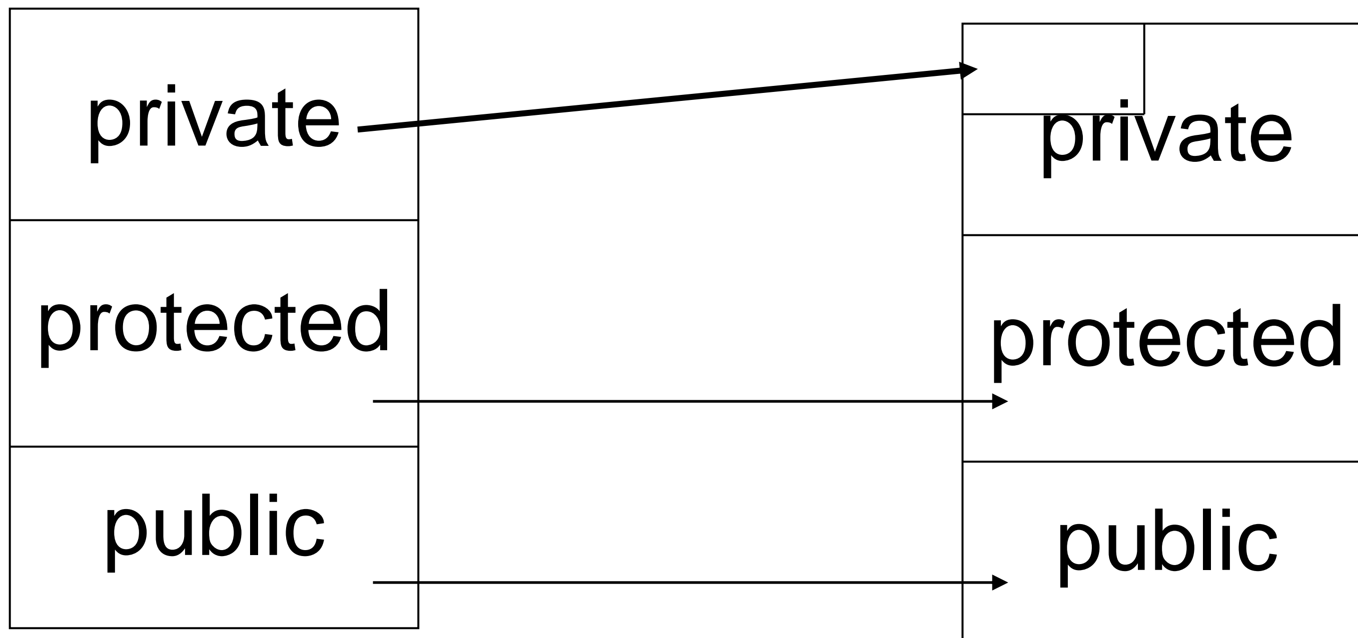
C++

- van többszörös öröklődés
- van polimorfizmus és dinamikus összekapcsolás – de csak mutatók és referenciák esetén!
- virtual kulcsszó a függvények előtt a dinamikus összekapcsolás támogatására

  ha nincs virtual kulcsszó, akkor
  A* pa = new B();
  pa->f() …
  esetén nem a B-beli f() hívódik meg, hanem az A-beli f() !
- az öröklődés minősítője lehet:
  public, protected, private

C++

- public öröklődés:



unoka?
külvilág?

C++

```
#include <iostream>
    using namespace std
class Vehicle {
    public:
        Vehicle()  {cout << "Vehicle Constructor" <<  \n;}

        virtual ~Vehicle() { cout << "Vehicle Destructor" << \n; }

        virtual void accelerate() {
                    cout << "Vehicle Accelerating" << \n;
                    }

        void setAcceleration(double a)  {acceleration = a;}
        double getAcceleration()  {return acceleration;}

    private:
          double acceleration;
    };
```

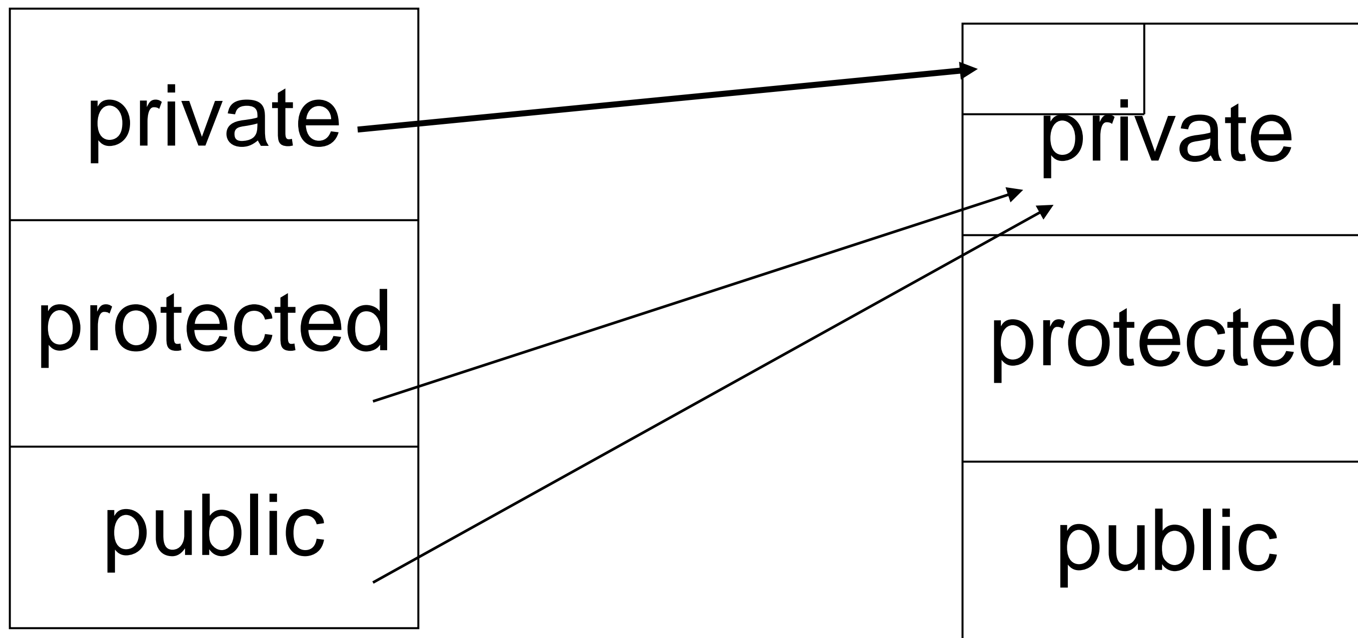
```
class Car: public Vehicle {
    public:
       Car() {cout << "Car Constructor" << \n;}
      virtual ~Car() {cout << "Car Destructor" << \n;}

       virtual void accelerate() {
                      cout << "Car Accelerating" << \n;
                      }
       void drive() {
                      cout << "Car Driving" << \n;
                      }

    private:
       // Car inherits acceleration accessors, member
    };
```

C++

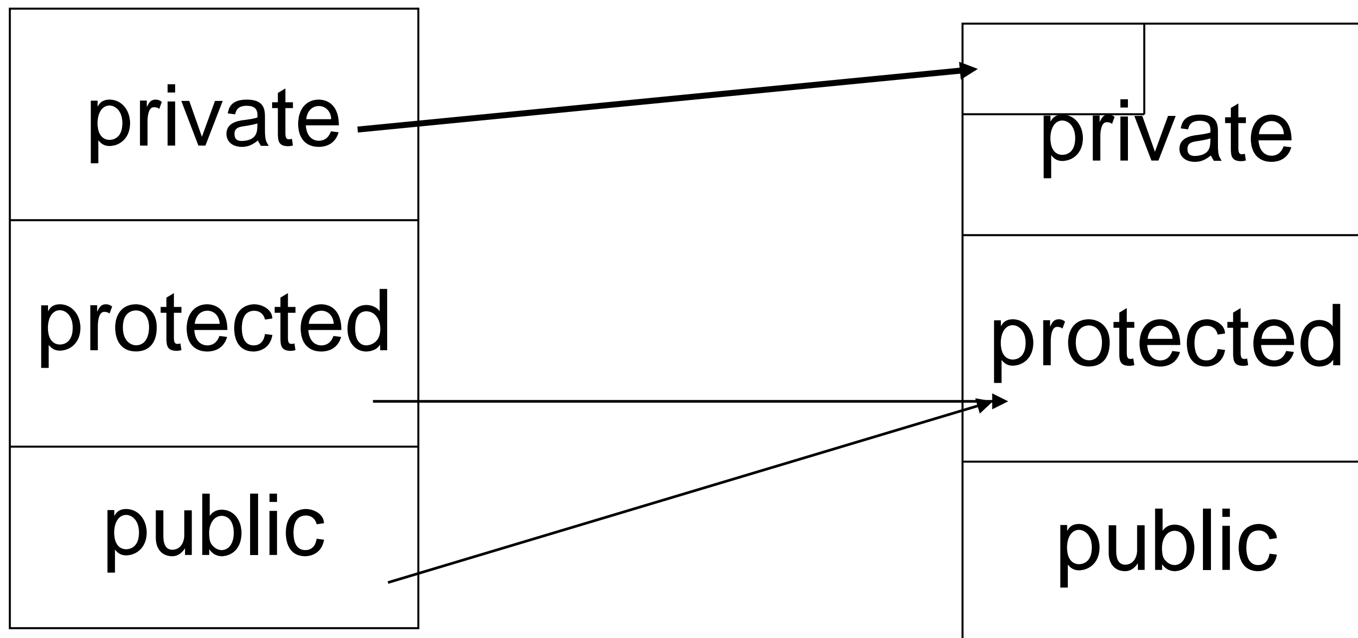
- private öröklődés:



unoka?
külvilág?

C++

- protected öröklődés:



unoka?
külvilág?

ős

leszármazott

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++

- nincs default ősosztály
- lehet (részlegesen) absztrakt osztályokat definiálni, ekkor az osztály nem példányosítható (pure virtual fv.)

```
virtual void draw() = 0;
```

C++

- **konstruktorok: “nincs objektum konstruktor nélkül”, ha kell, implicit hívódnak**
  - **neve megegyezik az osztály nevével**
  - **ha már megadtunk egy konstruktort, akkor default konstruktor nem definiálódik**
  - **a default konstruktor meghívja az attribútumok konstruktorát, de a beépített típusokat nem inicializálja (konzisztensen a C-vel)**
  - **a konstruktornak nem lehet visszatérési értéke**
- **a destruktort is explicite lehet hívni a delete operátorral, vagy implicit hívódik a blokkból való kilépéskor – fordított sorrendben**

C++

- Inicializáló lista: konstruktor hívásokhoz

```
class Coefficient
{  int myValue, myAccesses;
public:
        Coefficient(void)
        { myValue = 0; myAccesses = 0; }

        Coefficient(double initval)
        { myValue = initval; myAccesses = 0;
    }

        // ETC.
};
```

```
class StatusCoefficient : public Coefficient
{ Status myStatus;
public:
        StatusCoefficient(void)
        { myStatus = OverStatus; }

        StatusCoefficient(double initval,
                          Status initStatus)
        : Coefficient(initval)
        { myStatus = initStatus; }

        // ETC.
};
```

C++

Osztályváltozó definiálása és használata:

```
class Datum{
   int nap, ho, ev;
   static Datum alapert_datum;
 public:
   Datum(int nn=0, int hh=0, int ee=0); //...
   static void beallit_alapert(int,int,int);
 }
```

Mire jó? Pl., ha paraméter nélküli konstruktor hívás történik, akkor az alapert_datum értéket kapja meg az új objektum.

```
//Az előző konstruktor helyett:
 Datum::Datum(int nn, int hh, int ee){
   nap = nn ? nn : alapert_datum.nap;
   ho = hh ? hh : alapert_datum.ho;
   ev = ee ? ee : alapert_datum.ev;
 }
```

C++

```
void Datum::beallit_alapert(int n, int h, int e){
  //a statikus adattag értékének megváltoztatása
  Datum::alapert_datum = Datum(n,h,e);
}
```

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)

C++

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 belsejébe:

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
```

C++

- **Absztrakt osztály - öröklődés – dinamikus összekapcsolás**

```
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;

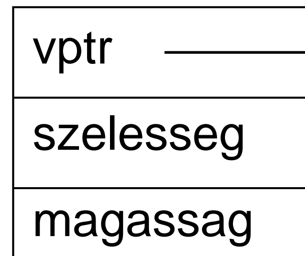
```
Lehet:  Teglalap t;
        Haromszog h;
  /*…*/
  Sikidom* a2 = &t;
  Sikidom* a3 = &h;
  …  a2->terulet(); //Teglalap::terulet()
  … a3->terulet(); //Haromszog::terulet()
```

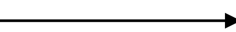
C++

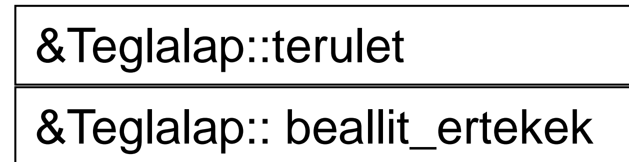
Megvalósítás:

Teglalap t;

Teglalap osztály virtuális metódustáblája







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 {
   …};
```

De: csak **egyszer** örököl mindent az Animal-től!

Object Pascal

- Többféle következetlenség
  - object – class
  - virtual (seb.) - dynamic (méret)
  - Láthatósági megkötések alkalmazhatóak – de fájlon belül nem érvényesek! – most már: strict private lehetősége 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! ☹

  abstract class – ezt szeretnék kiküszöbölni, itt már nem lehet …

Object Pascal

```
type
 TAnimal = class
  public
   constructor Create;
   function Verse: string ; virtual ;  // virtuális metódus
   function VerseStatic: string ;  // statikus metódus
  end ;
 TDog = class ( TAnimal)
  public
   constructor Create;
   function Verse: string ; override ;
     // felüldefiniáljuk a TAnimal virtuális metódusát
   function VerseStatic: string ;
     // felüldefiniáljuk a TAnimal statikus metódusát
  end ;
```

Object Pascal

A függvények megvalósítása:

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

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

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

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

Object Pascal

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

Object Pascal

```
ShowMessage(Animal2.VerseStatic);
   // a megjelenített ablak szövege: 'nem tudom nem tudom'
   // oka: a metódus statikus volt, és mi most egy TAnimal
   // típusú változó statikus metódusát hívtuk
ShowMessage( Dog1.VerseStatic);
   // a megjelenített ablak szövege: 'vau vau vau vau'
   // oka: most a TDog statikus metódusát hívtuk
Animal1:=Dog1;  // ez a polimorfizmus miatt  ok.
ShowMessage(Animal1.VerseStatic);
   // a megjelenített ablak szöveg: 'nem tudom nem tudom'
   // oka: a statikus metódust egy TAnimal 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
    // Define our polygon base class
    TPolygon = class
    private
      sideLength : Byte;
      sideCount  : Byte;
   // Define concrete methods - implemented in this parent class
      function  GetSideLength : Byte;
      procedure SetSideLength(length : Byte);
      // Define abstract methods - they must be implemented in a sub-class
      function  GetSideCount : Byte;       virtual; abstract;
      procedure SetSideCount(count : Byte); virtual; abstract;
      function  GetArea : Single;          virtual; abstract;
     published
      // Properties used to access private data fields
      // They use private methods to do this
      property length : Byte
         read GetSideLength       write SetSideLength;
      property count : Byte
         read GetSideCount     write SetSideCount;
      // The polygon constructor
      constructor Create(length : Byte);  virtual;    //!!
     end;
```

Object Pascal – absztrakt osztály

```
// Implement a concrete method of TPolygon
function TPolygon.GetSideLength : Byte;
begin
  Result := sideLength;    // Return the side length as currently set
end;

// Implement a concrete method of TPolygon
procedure TPolygon.SetSideLength(length : Byte);
begin
  sideLength := length;    // Set the side length from the passed parameter
end;

// Implement the TPolygon constructor
constructor TPolygon.Create(length : Byte);
begin
  // Save the passed parameter
  sideLength := length;
end;
```

Object Pascal – absztrakt osztály

```
// Define our triangle sub-class
 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

```
// Implement the triangle private methods
  function TTriangle.GetSideCount : Byte;
  begin
    Result := sideCount;
  end;

  procedure TTriangle.SetSideCount(count : Byte);
  begin
    sideCount := count;    // Set the side count from the passed parameter
  end;

  function TTriangle.GetArea : Single;
  begin
    // Return the area of the triangle
    Result := sideLength * sideLength * SQRT(3)/4;
  end;

  constructor TTriangle.Create(length : Byte);
  begin
    // Save the passed parameter
    sideLength := length;
     // And set the side count to 3 - a triangle
    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 – sealed osztály

- **sealed – nem származtathatő belőle**

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

- **fina**l – 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**
- **jelenleg az Apple Inc. fejleszti ! ☺**

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 és a reprezentáció egyben!
  - az implementációban pedig definiáljuk a metódusokat, ezzel tulajdonképpen az osztályt.

Objective-C

- Interfész:

```
@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

- implementáció:

```
#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
```

Objective-C

- láthatóság szabályozása:
  - @public,
  - @private,
  - @protected
  - @package direktívák
- alapértelmezett: @protected

Objective-C

- öröklődés – egyszeres, NSObject a legfelső szinten
- protocol –> = interface
- polimorfizmus, dinamikus kötés – automatikusan

```
@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.

```
#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 <u>isURL</u> 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. Pl.:

```
….
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) (még a beépített típusok is)
- nincs osztályon kívüli (globális) változó
- öröklődési fa gyökere az **Object** osztály
- nincs többszörös öröklődés – interfacek helyette

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 <u>immutable</u> osztályok
  - pl. boolean -> Boolean (nagybetű a különbség), de char ->Character és 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

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() {/*...*/}
  }
```

Eiffel

- többszörös öröklődés
- ismételt öröklődés
- a metódusok default virtuálisak
- van absztrakt osztály és metódus

Eiffel

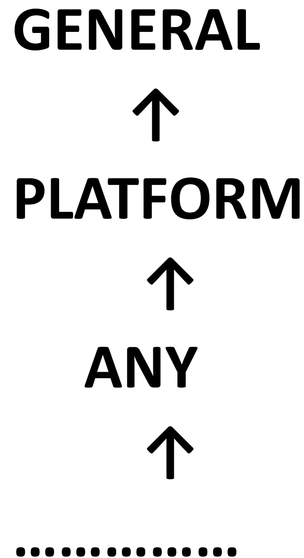
- a leszármazott mondja meg, hogy hogyan szeretné az örökölt attribútumokat és metódusokat felhasználni (anomália megoldása). Ez lehet:
  - **átnevezés (anomália) (rename kulcsszó)**
  - **export státusz megváltoztatása a származtatottban**
  - **metódus felüldefiniálása (pl. csak az előfeltételt, vagy az implementációt)**
  - **műveletek absztrakttá tétele (deferred) (késleltetett)**
  - **ismételt öröklődés (ismételt öröklődésnél csak egyet örököl, ill. megmondhatja, hogy legyen mindkettő, ha akarja)**

Eiffel – ismételt öröklődés

```
class D
  inherit D
    rename f as old_f
  end
  inherit D
     redefine f
     select f
  end
 feature
    f is
       do ...
          old_f; ...
     end -- f
end -- class D
```

Eiffel

- **select záradék**
- **osztály-hierarchia:**



+ NONE osztály

- **jellemzők láthatósága: megadható osztályonként:**

  feature{NONE}

  I:INTEGER

Eiffel - példa

```
deferred class KONTENER                -- absztrakt osztály

feature -- inicializáció

    make is
    do
        !!tartalom.make;
    ensure
        ures_tartalom: tartalom /= void and tartalom.empty
    end -- make

feature -- attribútumok

    tartalom : LINKED_LIST[KEZZELFOGHATO];

end -- class KONTENER
```

Eiffel - példa

```
deferred class KEZZELFOGHATO
feature -- initialization
    make(t : TORDELO; n : STRING) is
    require
        nem_void: t /= void and n /= void
    do …
    ensure
        beallitas_megtortent: tord = t and nev = n
        ures_hely: kornyezet = void and x = 0 and y = 0
    end -- make
feature -- metódusok
    move(ko : KONTENER; ux : INTEGER; uy : INTEGER) is
    require
      egyertelmu_ures_hely: ko = void implies ux = 0 and uy = 0
    do  …
    ensure
        beallitas_megtortent: kornyezet = ko and x = ux and y = uy
    end -- move
    cselekedj is
    do   …
    end -- cselekedj
```

Eiffel – példa

```
feature -- attribútumok
    tord : TORDELO;
    nev: STRING;
    kornyezet : KONTENER;
    x, y : INTEGER;
invariant
    konzisztens_tartalmazas: kornyezet /= void implies
            kornyezet.tartalom.has(current)
    extrem_poziciok_kizarva: x > -1000 and y > -1000 and x < 1000      and y
    < 1000
    egyertelmu_ures_hely: kornyezet = void implies x = 0 and y = 0
end -- class KEZZELFOGHATO
```

Eiffel

```
deferred class KEZZELFOGHATO_KONTENER
--Olyan dolgok őse, amelyek kézzelfoghatóak és konténerek egyben.
inherit
    KEZZELFOGHATO
        rename
            make as make_kezzelfoghato
        end -- rename
    KONTENER
        rename
            make as make_kontener
        end -- rename
feature -- inicialization
    make(t : TORDELO; n : STRING) is
    require
        nem_void: t /= void and n /= void
    do
        make_kezzelfoghato(t, n);
        make_kontener;
    ensure
        beallitas_megtortent: tord = t and nev = n
        ures_hely: kornyezet = void and x = 0 and y = 0
        ures_tartalom: tartalom /= void and tartalom.empty
    end -- make
end -- class KEZZELFOGHATO_KONTENER
```

Eiffel

```
class KUKA
inherit
    KEZZELFOGHATO_KONTENER
        redefine
            cselekedj
        end -- redefine
creation
    make
feature -- methods
    cselekedj is
    do
        if tartalom.empty
            then tord.tordel(nev + " üresen áll.%N");
            else tord.tordel(nev + " körül legyek szállldosnak.%N");
        end; -- if
    end;

end -- class KUKA
```

Python

- Érdekes, népszerű scriptnyelv
- Osztály definiálása:

```
class MyClass:
  „Egy egyszerű példa osztály”
  i = 42
  def f(x):
    return 'hello world!'
```

- Az osztályok is objektumok

Python

- Új adattagot bármikor bevezethetünk (!)

```
x = MyClass()
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

- és a végén töröljük az adattagot!

Python

- x.f() - ki fogja írni: hello world.
  Az objektum, mint első argumentum átadódik a függvénynek, x.f() ekvivalens MyClass.f(x) -szel.
- Konstruktor: ha létezik egy __init__() konstruktor (valójában: inicializáló) metódusa az osztálynak, akkor példányosításkor az objektum létrehozása után meghívódik, átadva a példányosításkor esetleg megadott paramétereket:

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
x = Complex(3.0,-4.5)
```

Python

(Többszörös) öröklődés

class DerivedClassName ([modulename.]Base1 [,[[modulename.]Base2,...]])

Ha egy hivatkozást nem talál az aktuális osztályban, akkor Base1-ben keresi. Ha ott sem, akkor Base1 őseiben. Ezután ha még mindig nem találta, akkor Base2-ben kezdi el keresni hasonlóan, és így tovább.

- Python 2.3-ban új szabályok a többszörös öröklődés feloldásánál !
- Érdemes megnézni:

http://www.python.org/download/releases/2.3/mro/

- lényegében: bejárja az összes őst, ha tudja, ha pedig nem tudja/nem lehet, akkor fordítási hibát ad
- (itt van egy közös ősosztály is már)

Ruby

- Teljesen objektumorientált
  - Minden típus osztály, így minden változó objektum
  - Az osztályok is objektumok
  - Az *Object* osztály minden más típusnak őse
  - Numerikus típusok hierarchiába szervezve,
    logikai típus nincs, a true és a false a TrueClass, illetve a *FalseClass* egyke osztályok egyetlen példánya.

- Jelölési konvenciók:
  - A nagybetűvel kezdődő változók konstansnak számítanak.
    (=> osztálynév nagybetűvel kezdődik)
  - A kisbetűvel kezdődő nevű változók lokálisak.
  - A tagváltozók @-cal kezdődnek.
  - A osztályszintű (statikus) változók @@-cal kezdődnek.
  - A globális változók $ jellel kezdődnek.

Ruby

class Vector def initialize(x, y) @x = x @y = y end end

A konstruktor az *initialize* nevű függvény. A Rubyban nincs függvénytúlterhelés, ezért egy osztálynak csak egy konstruktora lehet. Ha több konstruktorra van szükségünk, akkor ezeket más nevű függvényekként kell megvalósítanunk, és explicite kell őket meghívnunk az objektumra.
Az initialize függvény paraméterezése azonban nincs előre megszabva, így annyi paramétert kaphat, amennyit megszabunk.
Mivel a Ruby szemétgyűjtéses nyelv, destruktor nem létezik benne.

Ruby

A @ prefixszel ellátott változók a tagváltozók. Csakúgy, ahogy a lokális változókat nem kell definiálni, a tagváltozók is akkor jönnek létre, amikor először értéket kapnak.

A fenti példában tehát az osztálynak két tagváltozója van (@x és @y), amelyek a vektor koordinátáit tárolják. A tagváltozók nem láthatók kívülről, ezért az osztály ilyen formájában még nem használható. Készítsünk getter műveleteket a koordinátákhoz:

```
class Vector
    def initialize( x, y ) @x = x @y = y end
    def x @x end
    def y @y end
    end
```

Hozzuk létre a (2, 3) pont helyvektorát, és írjuk ki a koordinátáit!

```
v = Vector.new( 2, 3 ) puts v.x, v.y
```

Az eredmény: 2 3

Ruby

- Csak egyszeres öröklődés van, jelölése:

```
class ChildClass < ParentClass
    # …
end
```

- A virtualitás jelölésére nincs mód, de nincs is rá szükség, mert minden tagfüggvény virtuális.
- A tagváltozók privát hozzáférhetőségűek.
- A tagfüggvények hozzáférhetőségét befolyásolhatjuk a *private*, *public* és *protected* minősítőkkel. Az alapértelmezés a publikus.

Ruby

- Lehet objektumból (!) örököltetni:

```
class Foo …
end
  f = Foo.new
   g = Foo.new
 class << f
   def only_f
     puts "This works only for f.″
   end
end


f.only_f
```

A *g*-re nem lehet meghívni az *only_f*-et.
Az történik, hogy az örököltetéskor létrejön a *Foo* egy alosztálya, és az *f* dinamikusan megváltoztatja a típusát erre az osztályra.

Lehet így is:

```
def f.only_f
    # …
end
```

C#

- OOP támogatása nagyon hasonlít a Javáéhoz
  - Egyszeres öröklődés,
  - Interfészek (itt lehet többszörös öröklődés)
- Különbségekre példák

C#

- Lehetnek statikusan és dinamikusan kötött metódusok is:

```
class A {
   public void F() {
        Console.WriteLine("A.F");
   }
   public virtual void G() {
         Console.WriteLine("A.G");
   }
}
```

C#

```
class B: A {
    new public void F() {
        Console.WriteLine("B.F");
    }

    public override void G() {
        Console.WriteLine("B.G");
    }
}
```

C#

```
class Test  {
      static void Main()  {
            B b = new B();
             A a = b;
             a.F();
             b.F();
             a.G();
             b.G();
          }
 }
//A.F B.F B.G B.G
```

C#

- A *közvetlen ősosztályra* a **base** kulcsszóval hivatkozhatunk.
- A **sealed** kulcsszó használatával megakadályozhatjuk, hogy egy osztályból származtassanak vagy egy metódust felüldefiniáljon a származtatott osztály.
- Ha egy metódusnál a **sealed override** hozzáférést használjuk, akkor ezzel meggátoljuk, hogy egy származtatott osztályban felülírjuk ezt a metódust.
- Az **external** módosítóval rendelkező metódusok valamilyen más nyelven vannak implementálva. Éppen ezért a metódus törzsében csak egy pontosvessző áll. Az ilyen metódus nem lehet **abstract**.

C#

- Konstruktorok:
  - Közvetlenül a konstruktor törzsének végrehajtása előtt automatikusan történik egy másik konstruktor hívás is. Ezt nevezik konstruktor inicializálásnak is.
  - Kétfajta lehetőségünk van: vagy meghívjuk az ős valamelyik példány konstruktorát, vagy az adott osztály egy másik konstruktorát hívjuk meg először. Ha nem használjuk egyiket sem, akkor az ős alapértelmezett konstruktora hívódik meg.
  - Azaz a következő két deklaráció ekvivalens egymással:

```
C( ...) {...}
C( ...): base() {...}
```

C#

- Példák konstruktor inicializátorra:

```
class A {
        public A(int x, int y) {…}
  }
class B: A {
        public B(int x, int y): base(x + y, x - y) {…}
  }
class Text {
        public Text(): this(0, 0, null) {…}
        public Text(int x, int y): this(x, y, null) {…}
        public Text(int x, int y, string s)
        {// valami kód}
  }
```

C#

- Ha egy osztálynak nem deklarálunk semmilyen példány konstruktort, akkor egy olyan default konstruktor jön létre, ami az alapértelmezett konstruktor inicializátorral fog rendelkezni.
- Készíthetünk **private** konstruktorokat is, de ebben az esetben az osztály nem példányosítható és nem lehet örököltetni sem belőle.
  Akkor célszerű ezt használni, ha pl. csak statikus elemeket tartalmaz az osztályunk.
- **Statikus konstruktorok**
  A konstruktor neve előtt a **static** kulcsszót kell használni. A statikus konstruktorok az osztály inicializálásakor futnak le. Ez pontosan akkor van, amikor az osztály betöltődik. Ezekre a konstruktorokra nem lehet hivatkozni és természetesen nem is öröklődnek.

C#

- **Destruktorok**

  Az objektumok megsemmisülésekor hívódnak meg. Mivel a C#-ban is megvalósították az automatikus szemétgyűjtést, ezért a destruktorok automatikusan hívódnak meg, közvetlenül nem lehet őket hívni. A destruktorok az öröklődési láncon végighaladva egymás után hívódnak meg. (A szemétgyűjtő nyilvántartja azon objektumok listáját, amelyek osztályában definiáltunk destruktort.)

C#

- Szintaktikailag hasonlít a C++ destruktorhoz: ~MyClass(){}
- Voltaképpen csak a Finalize egy rövidített írásmódja. Ha ezt írjuk:

```
~MyClass() { // do work here }
```

a fordító ezt generálja:

```
protected override void Finalize() {
    try { // do work here }
    finally { base.Finalize(); }
}
```

C#

- Szemétgyűjtés felülbírálható az IDisposable interfész megvalósításával, ekkor kell egy Dispose() metódus.
  DE:

  Nem lehetünk biztosak benne, hogy a felhasználó megbízhatóan hívja…

  a using utasítást vezették be, hogy biztosan lefusson.

C#

Absztrakt osztályok

- Az **abstract** kulcsszó használatával azt jelezhetjük, hogy az adott osztály még nincs teljesen megvalósítva.
  - Absztrakt osztály emiatt nem példányosítható, csak ősosztály lehet.
  - Az absztraktként definiált metódusait a származtatott osztályban meg kell valósítani.
  - Természetesen egy absztrakt osztály nem lehet **sealed** (véglegesített).
  - Annak ellenére, hogy egy absztrakt metódus tulajdonképpen virtuális, nem használhatjuk a **virtual** megjelölést.
  - Természetesen **static** -ként sem lehet definiálni az absztrakt metódusokat.
  - Ezeket a metódusokat a származtatott osztályokban implementálhatjuk úgy, hogy az adott metódushoz megírjuk a törzs részt is.

C#

```
public abstract class Shape {
      public abstract void Paint
                              (Graphics g, Rectangle r);
}
public class Ellipse: Shape {
     public override void Paint
                              (Graphics g, Rectangle r) {
                                           g.drawEllipse(r); }
}
public class Box: Shape {
    public override void Paint
                              (Graphics g, Rectangle r) {
                                                 g.drawRect(r); }
 }
```

C#

- Egy metódus törzsében nem hivatkozhatunk az ős absztrakt metódusára:

```
class A {
        public abstract void F();
}
class B: A {
        public override void F()  {
            base.F();
         // Hiba, mert a base.F() metódus absztrakt
  }
}
```

C#

DE: ha nem a metódus, hanem az osztály absztrakt akkor lehet a függvénynek törzse. (Például a közös részeket az absztrakt osztályban meg lehet írni és csak a base kulcsszóval hivatkozunk rá.)

```
abstract class A0{
    public virtual void F() { System.Console.WriteLine("HELLO! "); }
    ….
}

class B0: A0
{
    public override void F()
    {
        base.F();
        // Nem hiba, hiába absztrakt az osztály
    }
}
```

C#

Az öröklődés: class Osztály: ŐsOsztály {..}

- Utód osztályban nem lehet bővebb hozzáférés, mint ős osztályban,
- a base kulcsszóval a közvetlen ősosztály metódusa elérhető.
- A virtuális függvényeknél a new kulcsszó elhagyásával warning-ot kapunk.

```
public class CA {
    public virtual void f() { }
}
public class CB : CA {
    public override void f() { } //felüldefiniálja
}
public class CC : CA {
    private new void f() { } //elrejti, nem virtuális függvénnyel
}
public class CD : CA {
    public new virtual void f() { } // új virtuális függvény (mostantól)!
}
public class CE : CA {
    public sealed override void f() { } //nem lehet többször átdefiniálni
}
```

C#

- Hasonlóan az absztrakt osztályoknál az öröklés:

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

public class C1 : A {
	public sealed override void f() { }
}

public sealed class C2 : A {
	public sealed override void f() { }
}
```

C#

- **Interfészek**
  - Nincs többszörös öröklődés C#-ban, helyette interface-k vannak.
  - Az interfész egy szerződés. Amelyik struktúra vagy osztály megvalósítja, annak meg kell felelnie ennek a szerződésnek.
  - Az interfész definiálhat metódusokat, indexelőket, property-ket, eseményeket, de azokat nem valósítja meg.
  - Egy interfésznek több más interfész is lehet őse, egy osztály ill. struktúra pedig több interfészt is megvalósíthat.
  - Az interfész jellegéből logikusan következik, hogy nem szabad használni az interfész tagjaira az **abstract**, **virtual**, **static**, **override** módosítókat.

C#

```
interface IMyInterface {
       void MethodToImplement();
    }


class InterfaceImplementer : IMyInterface {
    static void Main(){
        InterfaceImplementer iImp = new InterfaceImplementer();
        iImp.MethodToImplement();
       }

       public void MethodToImplement()
       {
          Console.WriteLine("MethodToImplement() called.");
       }
    }
```

C#

```
using System;

    interface IParentInterface {
        void ParentInterfaceMethod();
    }

    interface IMyInterface : IParentInterface {
        void MethodToImplement();
    }

    class InterfaceImplementer : IMyInterface {
        static void Main(){
            InterfaceImplementer iImp = new InterfaceImplementer();
            iImp.MethodToImplement();
            iImp.ParentInterfaceMethod();
        }

        public void MethodToImplement(){
            Console.WriteLine("MethodToImplement() called.");
        }

        public void ParentInterfaceMethod() {
            Console.WriteLine("ParentInterfaceMethod() called.");
        }
    }
```

C#

Interfészek megvalósítása:

```
interface I1
{
    void MyFunction();
}

interface I2
{
    void MyFunction();
}
```

C#

```
class Test : I1,I2
{
    public void MyFunction()
    {
        Console.WriteLine("Guess which interface I represent???!");
    }
}
```

C#

```
class AppClass1
  {
     public static void Main(string[] args)
     {
        Test t=new Test();
        I1 i1=(I1)t;
        i1.MyFunction();
        I2 i2=(I2)t;
        i2.MyFunction();
     }
  }
```

C#

De lehet <u>különböző</u> megvalósítást is adni rá:

```
class Test : I1,I2
  {
     void I1.MyFunction()
     {
       Console.WriteLine("Now I can say this here is I1 implemented!");
     }
     void I2.MyFunction()
     {
       Console.WriteLine("Now I can say this here is I2 implemented!");
     }
  }
```

C#

```
class AppClass2
  {
    public static void Main(string[] args)
    {
      Test t=new Test();
      I1 i1=(I1)t;
      i1.MyFunction();
      I2 i2=(I2)t;
      i2.MyFunction();
    }
  }
```

C#

- Lehetőség van operátor-túlterhelésre is, csak public static lehet!

Pl.

```
public static Digit operator+(Digit a, Digit b)
  {
      return new Digit(a.value + b.value);
  }
```

C#

- **Property**
  Definiálhatók *elérők* (accessor), melyek tulajdonságként viselkednek, de az írás (**set**) illetve olvasás (**get**) számára külön eljárások írhatók. Például, ha egy elérőre csak a get eljárást definiáljuk, akkor az elérő egy csak olvasható attribútumként fog viselkedni.

```
public class Button: Control
{
    private string caption;
    public string Caption
    {
        get { return caption; }
        set {
            if (caption != value)
            { caption = value; Repaint(); }
        }
    }
}
```

C#

- **Indexelők:**

Definiálhatók *indexelők* (indexer) is, melyekkel szögletes zárójellel indexelhető osztályok hozhatók létre. Fontos, hogy az indexelő paramétere bármilyen típus lehet.

Így bármilyen **object** úgy indexelhető, mint egy tömb.

Az indexelő deklarálásakor meg kell adni az elemek típusát, majd a **this**-t követi a formális paraméterlista szögletes zárójelben.

Ez a paraméterlista határozza meg az indexelés szintaxisát, benne nem szerepelhet **ref,** ill. **out** paraméter.

C#

```
class Grid {
    const int NumRows = 26;
    const int NumCols = 10;
    int[,] cells = new int[NumRows, NumCols];
    public int this[char c,int colm]  {
        get
        {
            c = Char.ToUpper(c);
              if (c < 'A' || c > 'Z') throw new       ArgumentException();
            if (colm < 0 || colm > = NumCols) throw new
                                                IndexOutOfRangeException();
            return cells[c - 'A', colm];
        }
        set
        {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') throw new ArgumentException();
            if (colm < 0  || colm > = NumCols) throw new
                                                IndexOutOfRangeException();
            cells[c - 'A', colm] = value;
        }
    }
}
```

C#

Példa:

```
namespace osztaly
{
    public abstract partial class Ősosztály
    {
        protected static int counter = 0; // static változók példányosítás nélkül is elérhetőek, az osztályobjektumhoz tartozó adattagok, de a példányokon keresztül is elérhetőek
        private string szoveg = "alap"; // inicializáló értékadás támogatott
        public string Szoveg // speciális adattag, értékadást és lekérdezést valósít meg
        {
            get
            {
                return this.szoveg;
            }
            set
            {
                this.szoveg = value;
            }
        }
    }
```

C#

```
namespace osztaly
{
    public abstract partial class Ősosztály    {
        protected Ősosztály(string szoveg) {
            this.szoveg = szoveg;
        }

        public static string milyenNapVan()   {
            return System.DateTime.Now.ToString("dddd");
        }

        protected abstract string szovegKerites();

        public virtual void konkatMaiNap()    {
            this.szoveg = System.String.Format(this.szovegKerites(), this.szoveg +
      Ősosztály.milyenNapVan());
        }
    }
}
```

C#

```
namespace osztaly
{
   public sealed class SzármaztatottOsztály : Ősosztály{
      public SzármaztatottOsztály() : base("Milyen szép nap van ma : ")
      // az ősosztály példányát konstans változóval paraméterezett konstruktorának meghívásával hozzuk létre
      {   ++Ősosztály.counter; // static adattag elérése az osztály objektumon keresztül
      }
      protected sealed override string szovegKerites() {
         return "--> {0} <--";
      }
      public sealed override void konkatMaiNap() {
         base.Szoveg = System.String.Format(this.szovegKerites(), base.Szoveg + Ősosztály.milyenNapVan() + " (" + System.DateTime.Now.ToString("d") + ")");
      // base - az ősosztály specifikációjának megfelelő adattagot, vagy metódust szeretnénk elérni,
      }
      public static int Counter {
         get
         {
            return Ősosztály.counter;
         }
      }
   }
}
```