



# Programozási nyelvek és módszerek

---

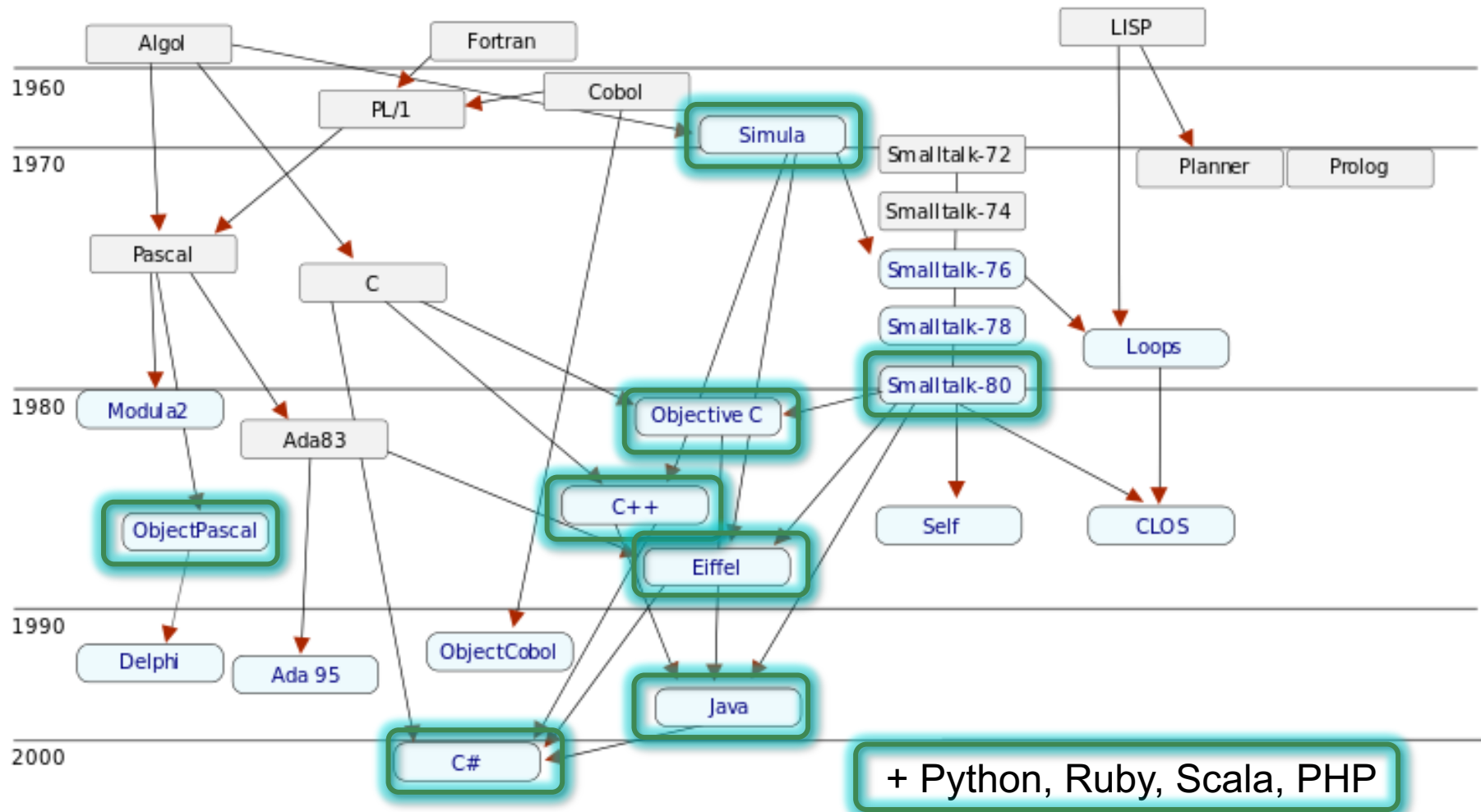
7. ELŐADÁS – OOP

# Mai óra

---

- OOP nyelvek jellemzői, szempontjai
- Programozási nyelvek OOP alapján
  - Simula 67
  - SmallTalk
  - C++ (előző előadás)
  - Object Pascal **NEW**
  - Objective-C
  - Java
  - Eiffel
  - Python
  - Ruby **NEW**
  - C#
  - Scala **NEW**
  - PHP **NEW**

# Fejlődés, hatások



# OOP fő elvei

---

OO moduláris struktúra

Adataabsztrakció

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 őssosztály?

- 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 létrehozasi param;
  INTEGER alapteruletX,alapteruletY;      comment par-ek típusa;
  VIRTUAL :                               comment virt fv megadása;
    PROCEDURE special_effects;
      ! Most jön az osztály 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 jönnek;
    ...
  INNER;
  ! Az inner rész jelöli a leszarmazotthoz tartozó specialis;
  ! utasitasokat. Ha nem lenne akkor az itteni utasitasok után;
  ! hajtodnanak vegre.;
  ! ide az jön amit meg ezután csinálni kell.
END epulet; ! Az END és a pontosvessző közötti rész 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

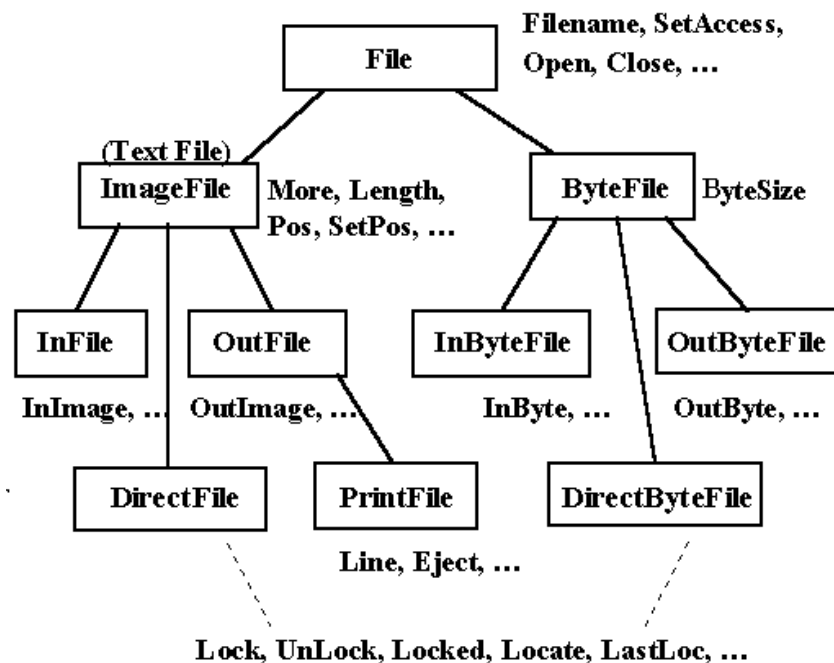
# Simula 67

## Láthatósági védelem

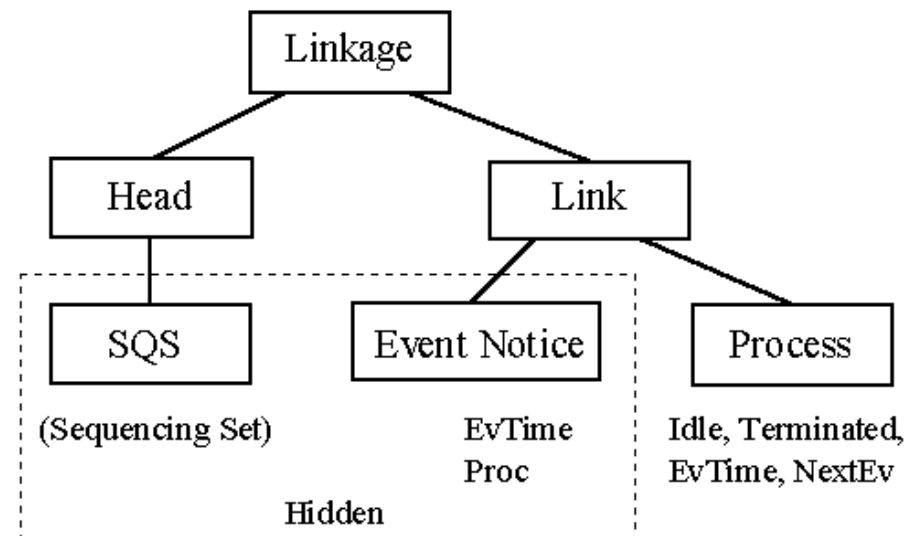
- hidden és protected prefixek az attribútumok előtt

## 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 típus nélküliek

- 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 kívülrre nem, viszont a metódusok globálisak
  - konvenció: a megjegyzésbe private-t írhatunk

Van osztályszintű attribútum és metódus

# SmallTalk

---

## Közös ősz osztály

- Object
  - objektum kiírása
  - objektum osztályának meghatározása
  - copy
  - klónozás

## Konstruktor

- osztálynak küldött üzenet

## Destruktor nincs

- szemétygyűjtéssel 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++

---

Előző előadás anyaga ...

# 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 T0s = class
    a, b: Integer;
    constructor Letrehoz(x, y: Integer);
    ...
end;
```

```
type TUj = class(T0s)
    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 T0s 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 8 – default interface bevezetése

---

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

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



# Java 8 – default interface bevezetése

---

```
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());
    }
}
```

# Eiffel

---

## Lehetőségek

- Többszörös öröklődés
- Ismételt öröklődés
- A metódusok default virtuálisak
- Van absztrakt osztály és metódus

# Eiffel

---

Lehet bevezetni kovariáns metódusokat az altípusban

Példa:

```
class Sielo
  feature szobatars (Sielo s)
end
class Sielo_Lany inherit Sielo
  redefine szobatars
  feature szobatars (Sielo_Lany g)
end
```

**Nem típusbiztos!**

Részleges megoldás

```
class Sielo
  feature szobatars (like Current)...
end
```

# Eiffel

---

osztály-hierarchia:

GENERAL



PLATFORM



ANY



+ NONE osztály  
feature{NONE}  
I: INTEGER

# Jellemzők láthatósága megadható osztályonként

---

```
class A
  feature {B, C}      ← B és C látja (és utódaik)
    X: INTEGER;
  feature {ANY}      ← public
    make(p_name : STRING ) is
      require
        p_name /= ""
      do
        name := p_name ...
  feature {NONE}      ← private
    Y: ARRAY[INTEGER];
end
```

# Eiffel

---

## rename

- átnevezés lehetősége

```
class D inherit
    C
        rename a as ac
    end
B
feature
...
end
```

---

Példa:

```
class ACCOUNT
creation
  make
feature
  balance: INTEGER
  owner: STRING
  number: STRING

  deposit( sum: INTEGER )
  require
    sum > 0
  do
    balance := balance + sum
  ensure
    balance = old balance + sum
  end -- deposit
```

---

```
withdraw( sum: INTEGER )  
require  
    sum > 0  
    sum <= balance  
do  
    balance := balance - sum  
ensure  
    balance = old balance - sum  
end - withdraw
```



---

```
feature
  make( owner_, number_: STRING )
    require
      valid_owner:    owner_ /= Void
      valid_number: number_ /= Void
    do
      owner  := owner_
      number := number_
    ensure
      owner = owner_ and number = number_
      balance = 0
    end -- make

invariant
  balance >= 0
  owner /= Void
  number /= Void
end --class ACCOUNT
```

---

```
class SAVINGS_ACCOUNT
inherit
    ACCOUNT
        rename
            make as account_make
        end
creation
    make
feature
    interest: INTEGER
    set_interest( interest_: INTEGER )
        require
            interest_ >= 0
        do
            interest := interest_
        ensure
            interest = interest_
        end -- set_interest
```

---

```
pay_interest
do
    deposit(balance*interest//100)
ensure
    balance = old (balance + balance*interest//100)
end
```

---

```
feature
  make( owner_, number_: STRING; interest_: INTEGER )
  require
    valid_owner:    owner_ /= Void
    valid_number:   number_ /= Void
    valid_interest: interest_ >= 0
  do
    account_make(owner, number_)
    set_interest(interest_)
  ensure
    owner = owner_ and number = number_
    balance = 0
    interest = interest_
  end -- make
invariant
  interest >= 0
end --class SAVINGS_ACCOUNT
```

# Eiffel – többszörös öröklődés

---

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

---

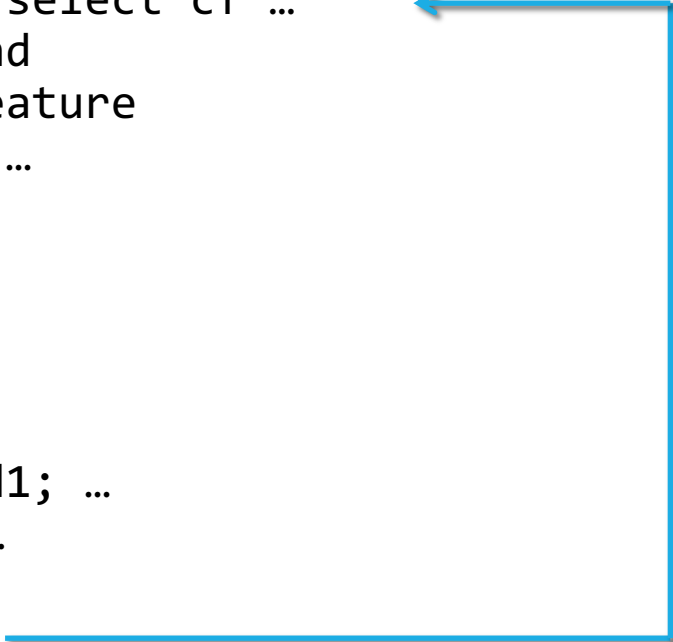
```
class A ...  
  feature  
    f ...  
end;
```

```
class B inherit A  
  redefine  
    f ...  
end;  
feature  
  f ...  
...  
end;
```

```
class C inherit A  
  redefine  
    f ...  
end;  
feature  
  f ...  
...  
end;
```

# Eiffel (select)

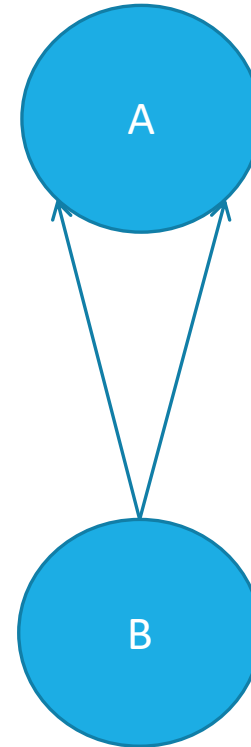
```
class D inherit
  B
    rename f as bf ...
  end;
  C
    rename f as cf
    select cf ...
  end
feature
  ...
end;
...
a1:A;
d1:D;
...
create d1; ...
d1.bf; ...
a1:=d1;
a1.f;
```



# Eiffel – ismételt öröklődés

```
class A
  feature
    f is ...
end -- class A
```

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





# Eiffel – példa

---

```
deferred class KONTENER -- absztrakt osztály
feature -- inicializáció
    make is
    do
        create 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;
  környezet : KONTENER;
  x, y : INTEGER;
invariant
  konzisztens_tartalmazas: környezet /= void implies
    környezet.tartalom.has(current)
  extrem_poziciok_kizarva: x > -1000 and y > -1000 and x < 1000
    and y < 1000
  egyertelmu_ures_hely: környezet = 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_kezzelfogható
        end -- rename
    KONTENER
        rename
            make as make_kontener
        end -- rename
```

# Eiffel

---

```
feature -- initialization
  make(t : TORDELO; n : STRING) is
    require
      nem_void: t /= void and n /= void
    do
      make_kezzelfogható(t, n);
      make_kontener;
    ensure
      beallitas_megtörtent: tord = t and nev = n
      ures_hely: kornyezeti = 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 legyenek szállidosnak.%N");
    end; -- if
  end;
end -- class KUKA
```

# Python

---

## Osztály definiálása

```
◦ class MyClass:  
    """Egy egyszerű példa osztály"""  
    i = 42  
    def f(self, 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`

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

---

Minden függvény, ami egy osztály attribútuma, az osztály példányainak egy metódusa.

- Ehhez nem szükséges, hogy a függvény definíció szövege az osztály definícióján belül legyen, elég egy értékadás az osztály egy lokális változójának...

```
def f1(self, x, y):  
    return min(x, x+y)  
  
class C:  
    f = f1  
    def g(self):  
        return 'hello world'  
    h = g
```

C példányainak 3 metódusa van: f,g,h

# Python – osztály és példányváltozók

---

```
class Dog:
    kind = 'canine'    # osztály változó, minden példányra azonos
    def __init__(self, name):
        self.name = name    # példányváltozó, egyedi minden
példányra
```

```
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                # minden kutyára
'canine'
>>> e.kind                # minden kutyára
'canine'
>>> d.name                # egyedi az objektumra
'Fido'
>>> e.name                # egyedi az objektumra
'Buddy'
```

# Python

---

## (Többszörös) öröklődés

- `class DerivedClassName (Base1 [,Base2,...])`

### „old style”:

- 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

### „new style”:

- Más szabályok – ahogy a példa mutatja

# Python

---

```
class A:
    def m(self):
        print("m of A called")
class B(A):
    def m(self):
        print("m of B called")
class C(A):
    def m(self):
        print("m of C called")
class D(B,C):
    pass
```

Ha az m-t a D –beli x-re hívjuk: x.m(), akkor az "m of B called" lesz az eredmény.

Ha megcseréljük a D-ben a sorrendet: „class D(C,B):„-re, akkor az "m of C called" lesz az eredmény.

# Python

---

Ha csak az egyik leszármazott definiálja felül az m-t:

```
class A:
    def m(self):
        print("m of A called")
class B(A):
    pass
class C(A):
    def m(self):
        print("m of C called")
class D(B,C):
    pass
X = D()
x.m()
```

# Python

---

Két python fordítóval:

- `$ python diamond1.py`
  - m of A called
- `$ python3 diamond1.py`
  - m of C called

Ahhoz, hogy ugyanaz legyen a viselkedése Python2-ben, mint Python3-ban, az kell, hogy az osztály az object-től örököljön

- `class A(object)`

# 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 egyike osztályok egyetlen példánya.



# Ruby

---

## 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 függvényterhelés, ezért egy osztálynak csak egy konstruktora lehet.
- Ha több konstruktorra van szükségünk, akkor ezeket más nevű függvényekként kell megvalósítanunk, és explicité kell őket meghívunk az objektumra.
- Az initialize függvény paraméterezése azonban nincs előre megszabva, így annyi paramétert kaphat, amennyit megszabunk.

Mivel a Ruby szemétygyűjtéses 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

# Ruby

---

```
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

---

```
class Person
  def initialize(nev)
    @nev = nev
  end
end
class Employee < Person
  def initialize(nev, fizetes)
    super(nev)
    @fizetes = fizetes
  end
end
```

Származtatásnál csak a metódusok öröklődnek, az adattagok NEM, mivel azok mindig objektumokhoz vannak kötve.

# Ruby

---

Lehet példányból dinamikusan örököltetni, és módosítani

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

# Ruby

---

A g-re nem lehet meghívni az only\_f-et.  
Az történik, hogy az örököltetéskor létrejön a Foo egy alosztálya, és az f dinamikusán megváltoztatja a típusát erre az osztályra.

Lehet így is

```
◦ def f.only_f  
    # ...  
end
```



# Ruby

---

## Modulok: a többszörös öröklődés helyett

- Modul neve nagybetűvel kezdődik
- Nem példányosítható, az osztályok modulokat foglalhatnak magukba
- A befogadott modul metódusai az osztály példánymetódusai lesznek
- Mixineknek nevezik
- A modulok tartalmazhatnak metódusokat, állandókat, más modulokat, vagy osztályokat
- Örökölhetnek (include) más moduloktól, de osztályoktól nem

# Ruby

---

```
module A
  def a
    puts "A1"
  end
end
module B
  def b
    puts "B1"
  end
end
class Test
  include A
  include B
end
obj = Test.new
obj.a      # => A1
obj.b      # => B1
```

# Ruby

---

```
module Debug
  def whoAmI?
    "#{self.type.name} (\##{self.id}): #{self.to_s}"
  end
end
class Phonograph
  include Debug
  # ...
end
class EightTrack
  include Debug
  # ...
end
```

# Ruby

---

```
ph = Phonograph.new("West End Blues")
et = EightTrack.new("Surrealistic Pillow")
ph.whoAmI?
» "Phonograph (#537766170): West End Blues"
et.whoAmI?
» "EightTrack (#537765860): Surrealistic Pillow"
```

# 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() {  
        // Elfedés  
        Console.WriteLine("B.F");  
    }  
  
    public override void G() {  
        // Felüldefiniálás  
        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 őosztályra 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`.

## 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 őt valamelyik példány konstruktorát, vagy az adott osztály egy másik konstruktorát hívjuk meg először.
  - Ha nem használjuk egyiket sem, akkor az őt 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 például 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.

## 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 felülbíráható 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 lefusszon.



# 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 ősz absztrakt metódusára:

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

# 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() {
        // Nem hiba, hiába absztrakt az osztály
        base.F();
    }
}
```

# C#

---

Az öröklődés: `class Osztály: ŐsOsztály {..}`

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

# C#

---

```
public class CA {  
    public virtual void f() { }  
}  
  
public class CB : CA {  
    public override void f() { } //felüldefiniálja  
}  
  
public class CC : CA {  
    private new void f() { } //elrejtí, nem virtuális függvénnel  
}  
  
public class CD : CA {  
    public new virtual void f() { } // új virtuális függvény!  
}  
  
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, propertyket, 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#

---

```
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("Which interface?");
    }
}
```

# 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();
    }
}
```

Tetszőleges statikus típus esetén az egyetlen implementáció fut le.

# C#

---

De lehet különböző megvalósítást is adni rá:

```
class Test : I1,I2
{
    void I1.MyFunction()
    {
        Console.WriteLine("I1 implemented!");
    }

    void I2.MyFunction()
    {
        Console.WriteLine("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();
    }
}
```

Az egyes implementációknak megfelelően fut le.

Mi történik, ha a `t.MyFunction`-t hívjuk?

- Hiba ... nincs ilyen implementáció.

# 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.



# C#

---

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

## 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 SzarmaztatottOsztaly : Ősosztaly{

        public SzarmaztatottOsztaly() : base("Milyen szép nap van ma : ")
        // az Ősosztaly példányát konstans változóval paraméterezett
        // konstruktorának meghívásával hozzuk létre
        {
            ++Ősosztaly.counter;
            // static adattag elérése az osztály objektumon keresztül
        }

        protected sealed override string szovegKerites() {
            return "--> {0} <--";
        }
    }
}
```

# C#

---

```
public sealed override void konkatMaiNap() {
    base.Szoveg = System.String.Format(this.szovegKerites(),
        base.Szoveg + Őosztály.milyenNapVan() + " (" +
        System.DateTime.Now.ToString("d") + ")");
    // base - az őosztály specifikációjának megfelelő
    // adattagot, vagy metódust szeretnénk elérni,
}

public static int Counter {
    get
    {
        return Őosztály.counter;
    }
}
}
```

# Scala

---

```
class Point(xc: Int, yc: Int) {  
    var x: Int = xc  
    var y: Int = yc  
    def move(dx: Int, dy: Int) {  
        x = x + dx  
        y = y + dy  
    }  
    override def toString():  
        String = "(" + x + ", " + y + ")";  
}  
val pt = new Point(1,2)
```



# Scala

---

Ha object-et írunk class helyett, akkor singleton objektum lesz, nem kell konstruktor sem:

```
object Point extends App {  
    var x: Int = 0;  
    var y: Int = 0;  
    def setXY(xp: Int, yp: Int) {  
        x = xp;  
        y = yp;  
    }  
    def move(dx: Int, dy: Int) {  
        x = x + dx  
        y = y + dy  
    }  
    override def toString(): String =  
        "(" + x + ", " + y + ")";  
    setXY(2, 3);  
    println(toString());  
}
```

# Scala

---

## Öröklődés:

```
abstract class Element {
  def contents: Array[String]
  val height = contents.length
  val width =
    if (height == 0) 0 else contents(0).length
  def demo() {
    println("Element's implementation invoked")
  }
}
class ArrayElement (val contents: Array[String] )
  extends Element {
  override def demo() {
    println("ArrayElement's implementation invoked")
  }
}
```

# Scala

---

Van lehetőség „trait”-ekre

```
trait Quacking {  
  def quack() = {  
    println("Quack quack quack!")  
  }  
}  
class Duck {  
  val name = "Kacsa"  
}  
val d = new Duck with Quacking  
d.quack
```

# Scala

---

trait – nagyon hasonlít az osztályhoz - még adattag is lehet benne!

```
trait Book {  
    def title:String  
    def title_=(n:String):Unit  
    def computePrice = title.length * 10  
}
```

Ennek is az AnyRef az őse.

# Scala

---

Lehet belőle osztályt is származtatni és with kulcsszóval hozzátenni

Különbség osztály és trait között:

- trait-nek nem lehet argumentuma
- super hívás értelmezése dinamikusan, linearizációval – ez segít a többszörös rombusz öröklődés problémájánál

# Scala

---

Többszörös öröklődés – linearizáció:

```
class Foo extends Bar with A with B with C {  
  // implementation...  
}
```

Egy 'super' hívás esetén a keresés/feldolgozás sorrendje:

- Foo -> C -> B -> A -> Bar -> AnyRef -> Any

# PHP

---

PHP 5-től van rendes támogatás

- újraírták az egészet

Egyszeres öröklődés

- 5.4-től trait-ek!

Interfészek

public, protected, private láthatóságok

- Alapértelmezett a public

# PHP

---

```
class osztaly{
    public $attr1;
    protected $attr2 = 2;
    private $attr3;

    public function setAttr3($attr3){
        $this->attr3 = $attr3;
    }

    function kiir(){
        echo $this->attr1." ".
            $this->attr2." ".$this->attr3;
    }
}
```

```
$peldany = new osztaly();
```

```
$peldany->attr1 = 5;
$peldany->setAttr3(3);
```

```
$peldany->kiir();
```

Eredmény: 5 2 3



# PHP

---

```
class a {  
    public $attr1;  
}
```

```
class b extends a{  
}  
$b = new b();  
$b->attr1 = 4;
```

Egy osztály nem valósíthat meg 2  
interface-t amik azonos  
függvényneveket tartalmaznak!

```
interface I {  
    public function do1();  
}  
interface I2 extends I{  
    public function do2();  
}  
class c extends b implements I2 {  
    public function do1(){  
        (...)  
    }  
    public function do2(){  
        (...)  
    }  
}
```

# PHP

---

Absztrakt osztályok és függvények hasonlóan mint máshol:

```
abstract class ab {  
    abstract protected function getVal();  
  
    public function printIt(){  
        print $this->getVal();  
    }  
}
```

# PHP

---

```
class Foo {
    public static $statvar = "foo";

    public function staticValue(){
        return self::$statvar;
    }
}

class Bar extends Foo {
    public function fooStatic(){
        return parent::$statvar;
    }
}
```

```
print Foo::$statvar;

$foo = new Foo();
print $foo->staticValue();
print $foo->statvar; // Undefined
property!

print $foo::$statvar;
$classname = "Foo";
print $classname::$statvar; //5.3-tól!

Bar::$statvar;
$bar = new Bar();
$bar->fooStatic();
```

Statikus esetben nincs \$this, helyette self és parent

# PHP

---

## Trait

- Kód újrafelhasználás a cél.
- Metódusok egybefoglalása, akár törzzsel is...
- Nem lehet önállóan példányosítani.

```
trait a{  
    function a1() { ... }  
    function a2() { ... }  
}
```

```
class b {  
    use a;  
    ...  
}
```

# PHP

---

```
trait HelloWorld {  
    public function sayHello() {  
        echo 'Hello World!';  
    }  
}  
  
class TheWorldIsNotEnough {  
    use HelloWorld;  
    public function sayHello() {  
        echo 'Hello Universe!';  
    }  
}  
  
$o = new TheWorldIsNotEnough();  
$o->sayHello(); //Hello Universe!
```

# PHP

---

## Precedencia:

- Felülírja az őstől örökölt metódust, de a „parent”-tel a traitből lehet hivatkozni az őstre.
- Az osztály metódus felülírja a trait-ből kapottat. Több is használható egyszerre, vesszővel tagolva, de ha kettő azonos nevű metódust vezet be, az hibát okoz.
- Az „as”-al a metódus láthatóságát is lehet változtatni vagy aliasozni akár

```
use HelloWorld { sayHello as private myPrivateHello; }
```

# PHP

---

## Traiteknél precedenciák:

- Abban az esetben, ha egy osztályon belül van egy olyan származtatott tagunk, aminek a neve megegyezik egy, az osztályon belül használt traiten belüli tag nevével, akkor a traiten belüli tag felülírja az osztály származtatott tagját.

# PHP

---

```
class Base {  
    public function sayHello() { echo 'Hello '; }  
}  
trait SayWorld {  
    public function sayHello() {  
        parent::sayHello(); echo 'World!';  
    }  
}  
class MyHelloWorld extends Base { use SayWorld; }  
$o = new MyHelloWorld(); $o->sayHello();
```

Itt a kimenet Hello World! lesz.



# PHP

---

Ha

- egy osztályon belüli tag neve megegyezik egy, az osztályon belül használt traiten belüli tag nevével,
- de ez a tag nem származtatott tag,

akkor az osztály felülírja a traiten belüli tag viselkedését

# PHP

---

```
trait HelloWorld {  
    public function sayHello() {  
        echo 'Hello World!';  
    }  
}  
  
class TheWorldIsNotEnough { use HelloWorld;  
    public function sayHello() {  
        echo 'Hello Universe!';  
    }  
}  
  
$o = new TheWorldIsNotEnough();  
$o->sayHello();
```

Itt a kimenet Hello Universe! lesz és nem Hello World!

# PHP

---

Egy osztály több traitet is fel tud használni,  
névütközés esetén a lehetőségek:

```
trait A {  
    public function smallTalk() { echo 'a'; }  
    public function bigTalk() { echo 'A'; }  
}  
trait B {  
    public function smallTalk() { echo 'b'; }  
    public function bigTalk() { echo 'B'; }  
}
```

# PHP

---

```
class Talker {  
    use A, B {  
        B::smallTalk insteadof A;  
        A::bigTalk insteadof B;  
    }  
}  
  
class Aliased_Talker {  
    use A, B {  
        B::smallTalk insteadof A;  
        A::bigTalk insteadof B;  
        B::bigTalk as talk;  
    }  
}
```

# PHP

---

Traitektől kapott láthatóság változtatható:

```
trait HelloWorld {  
    public function sayHello() {  
        echo 'Hello World!';  
    }  
}  
// Change visibility of sayHello  
class MyClass1 {  
    use HelloWorld { sayHello as protected; }  
}  
// Alias method with changed visibility  
// sayHello visibility not changed  
class MyClass2 {  
    use HelloWorld { sayHello as private  
myPrivateHello; }  
}
```

# PHP

---

Traitek egymásba ágyazhatók:

```
trait Hello {  
    public function sayHello() { echo 'Hello ';  
}  
}
```

```
trait World {  
    public function sayWorld() { echo 'World!'; }  
}
```

# PHP

---

Traitek egymásba ágyazhatók:

```
trait HelloWorld {  
    use Hello, World;  
}  
class MyHelloWorld {  
    use HelloWorld;  
}  
$o = new MyHelloWorld();  
$o->sayHello();  
$o->sayWorld(); ... az output Hello World!
```

# PHP

---

Traitekben lehetnek absztrakt metódusok is:

```
trait Hello {  
    public function sayHelloWorld() {  
        echo 'Hello' . $this->getWorld();  
    }  
    abstract public function getWorld();  
}
```



# PHP

---

Traitekben lehetnek absztrakt metódusok is:

```
class MyHelloWorld {  
    private $world;  
    use Hello;  
    public function getWorld() {  
        return $this->world;  
    }  
    public function setWorld($val) {  
        $this->world = $val;  
    }  
}
```

# PHP

---

Traitekben lehetnek statikus adattagok és metódusok is, statikus adattag csak metóduson belül, statikus metódus osztálymetódusként viselkedik:

```
trait StaticExample {  
    public static function doSomething() {  
        return 'Doing something';  
    }  
}  
  
class Example {  
    use StaticExample;  
}  
  
Example::doSomething();
```