

# OOP – kezdő kockáknak

az OOP mozaikszó az object-oriented programming szavakat rejti, ami magyarul objektum orientált programozást jelent

Most hagyjuk az elméletet egyelőre. Kezdjünk kóddal:

```
package
import classes.Box;
import classes.SmallBox;
import classes.Point;

public class Main

    public function Main()
        init();

    private function init():void
        var bx:Box = new Box( this, new Point() );

        var bx2:Box = new Box( this, new Point(60, 0) );
        bx2.color = 0xff0080;

        var sbx:SmallBox = new SmallBox( this, new Point(120, 0) );
```

```
package classes

public class Box
    private var box_instance:Box;

    public function Box( aParent:Panel, aPoint:Point, aColor:int = 0 )
        box_instance = this;
        box_instance.x = aPoint.x;
        box_instance.y = aPoint.y;
        box_instance.draw( aColor );
        aParent.addChild( box_instance );

    private function draw( aColor:int ):void
        drawRectangle( 0, 0, 50, 50, aColor );

    public function get box():Box
        return box_instance;

    public function set box( aValue:Box ):void
        box_instance = aValue;

    public function set color( aValue:int ):void
        draw( aValue );
```

```
package classes

public class SmallBox extends Box
    public function SmallBox(aParent:Panel, aPoint:Point, aColor:int=0)
        super(aParent, aPoint, aColor);

    override protected function draw( aColor:int ):void
        drawRectangle( 0, 0, 30, 30, aColor );
```

package classes

```
public class Point
public var point:Object;
private var _x:Number;
private var _y:Number;

public function Point( aX:Number = 0, aY:Number = 0 )
    _x = aX;
    _y = aY;
    point = { x:_x, y:_y };

public function toString():String
    return "Point x:" + _x + " Point y:" + _y;

public function get x():Number
    return _x;

public function set x( aValue:Number ):void
    if ( _x != aValue )
        _x = aValue;

public function get y():Number
    return _y;

public function set y( aValue:Number ):void
    if ( _y != aValue )
        _y = aValue;
```

A kód [pszeudokód](#) és feltételezzük hogy minden a terv szerint működik. Az alapértelmezett szín - ha nincs megadva semmi, vagy az érték nulla - a fekete. Mi lesz az output? Nem beugratós a kérdés.

Megoldás: [link](#)

Az igazi cél nyilván nem a jó megfejtése volt a kérdésnek (bár az is!), hanem a kód tanulmányozása.

Első ránézésre talán a tagolást vesszük észre, az egységeket, meg az ismétlődő mintákat. Lássuk:

A kód négy nagyobb egységre bontható, mindegyik egység tartalmazza a „package” és a „class” szavakat az egység elején. A tabolás segítségével különülnek el jobban a kisebb egységek, meg persze a kihagyott sorokkal. A „package”-n kívül még három, egység eleji „p” betűs szó van: public, protected és private (nyilvános, védett, magán). A kisebb egységek jórészt függvények (function) - ezek szerint valamiféle munkát végeznek –, vagy adatok, amiket a függvények használnak. Aki tud angolul egy keveset, előnyben van, mivel ismeri jópár szó jelentését.

/this:ez, class:osztály, package:csomag, box:doboz, main:fő, point:pont, small:kicsi, instance:példány, if:ha, return:visszatér, new:új, override:felülír, stb./

Vajon mi ez a négy egység, ami függvényekre és adatokra bontható, és tele van „p” betűs szavakkal?

# Egy kis elmélet

Először beszéljünk a „mihez képest”-ekről: miért kapott az objektum orientált szemlélet egy külön fejezetet a programozás elméletben?

Legalább egy ellenpéldát érdemes megemlíteni a teljesebb kép kedvéért, a procedurális programozási módszert.

Gyors megközelítésben: ha egy programozási feladat megoldását alprogramokból építjük fel, amik egymástól többé-kevésbé függetlenek megjelenítésben és működésben is, procedurális programozásról beszélhetünk. A procedura műszó nem ördögöség, csak egy változata a különböző programozási nyelvek erre a programozási eszközre használt kifejezéseinek: függvény (function), metódus (method), rutin (routine), eljárás (procedure), stb. – a lényegi munkát végző egységek.

Egy procedurális programkód [változókra](#) (variables), [adatszerkezetekre](#) (data structures), [eljárásokra](#) vagy [függvényekre](#) (procedures, functions) bontható, míg egy objektum orientált kód [objektumokból](#) áll, azok viselkedését (metódusok) és adatait (attributes) tartalmazza.

A legfontosabb különbség a két szemlélet között, hogy amíg a procedurális programozás eljárásokat használ arra, hogy dolgozzon adatszerkezeteken, addig az objektum orientált programozás összeköti a kellemest a hasznossal, így egy objektum leggyakrabban a saját adatain és adatstruktúráin dolgozhat a saját metódusaiban.

## Példák:

```
procedure Udvozles( Nev: String );
begin
  Writeln( 'Üdvözöllek, ',Nev,', a klubban!' );
end;

var
  Nev: String;
begin
  Nev := "jck";
  Udvozles( Nev )
end.
```

```
public class Osztaly
{
  var nev:String;

  public function Osztaly()
  {
  }

  public function udvozles(n:String):void
  {
    nev = n;
    console( 'Üdvözöllek ' + nev + ' a klubban!' );
  }
}

...
var peldany: Osztaly = new Osztaly();
peldany.udvozles("jck");
```

# Az osztály fogalma

Egy osztály felfogható egy jól záródó doboznak, amelynek csak a publikusnak definiált részeihez férhet hozzá a külvilág. Ezek a „részek” [attribútumok](#) (attributes) és [metódusok](#) (methods) lehetnek. Osztályt általában minden programozási nyelven a „class” kulcsszóval hozunk létre. Egy osztály elkészített példánya az objektum, amiről a módszer a nevét kapta.

/Általában az osztályokban meghatározzuk hogyan készüljenek el a példányok, nem bízunk a dolgot a véletlenre. Erre az osztály konstruktorát (constructor) használjuk, ami az osztály nevét viselő metódus. Az osztály konstruktora minden esetben lefut, ha (általában egy „new” paranccsal) létrehozunk egy példányt. Ha nem írtunk konstruktort, a programnyelv biztosít egy alapértelmezettet az osztálynak.

Egy osztályt maradéktalanul meghatároz a neve és a csomag/csomagok nevével leírt útvonal. Így hivatkozhatunk rá a kódban is. (pl. classes.Point)

## Forró vizet a kopaszra!

Egy ennyire eltérő módszer megtanulása nem csak egy felesleges pipa egy listában? Minek érné meg telerakni a kódot egy rakat kifejezéssel, amik jócskán megnövelik a forráskód hosszát? Miért hoz ez bármi pluszt egy procedurális megközelítéssel szemben?

Igazából nem pluszt, hanem mást hoz. Pár számolási művelet kedvéért tényleg nem érdemes osztálystruktúrát létrehozni. Az [OOP](#) erőssége a nagyobb bonyolultságú feladatoknál domborodik csak ki.

Van pár fogalom, amit érdemes már az elején tisztázni, mivel ezek mondhatni az alappillérei minden OO nyelvnek:

egységbe zárás (**encapsulation**), öröklődés (**inheritance**), többalakúság (**polymorphism**).

## Encapsulation

Összehasonlításként elmondhatjuk, hogy a procedurális nyelvekben a globális és lokális változók/függvények felelnek meg egy OOP nyelv példányaira jellemző tulajdonságoknak. Hagyományosan a globális változókat az egész programkódból elérhetjük, míg a lokális változók csak a program bizonyos részeiből olvashatók és módosíthatók.

Az OOP-ben ennél jóval összetettebb láthatósági lehetőségekkel operálhatunk, amikkel például eleve elrejtethető a kód kiegészítő műveleteinek sora, vagy lehetetlenné tehető egy fontos tulajdonság

kívülről (más látókörből /*scope*/, másik osztályból, stb.) való módosítása. Így átláthatóbbá és hatékonyabbá tehetjük a forráskódot.

Egy létrehozott példány tulajdonságai mindig csak az adott példányra jellemzőek, mivel minden példány rendelkezhet saját, az ugyanabból az osztályból létrehozott példányoktól eltérő, egyedileg módosított tulajdonságokkal is. A tulajdonságokat szükségtől függően kívülről is láthatóvá tehetjük, de jellemzően inkább elrejtük, egységbe („kapszulába”) zárjuk őket.

Általában az OOP nyelvek három-, vagy négyféle fő láthatóságot módosító kulcsszót (access modifiers) használnak. Ezek a **private** (láthatatlan), **protected** (részben látható), **public** (látható) és néhány nyelvben az **internal** (részben látható). Egy tulajdonsághoz (attribútum, metódus) való hozzáférés (írás, olvasás) tovább módosítható a **getter/setter** függvényekkel.

## Inheritance

Az öröklődéssel az osztályok összedobált halmazába egy további strukturálási lehetőséget vihetünk.

Egy elkészült osztályból származtatható egy újabb. A gyerek osztály megfelel ősenek (örökli a szülő minden tulajdonságát), de tovább bővítheti a tulajdonságok számát, vagy specializálhatja a megörökölt tulajdonságokat. Sajátjaként kezeli az érintetlenül hagyott tagokat is, tehát egy gyerekből elérhető a szülő minden egyes tulajdonsága akkor is, ha a kódban ez nem jelenik meg explicit módon. Egy gyereknek egy, vagy több szülője lehet, ez programozási nyelvtől függő.

## Polymorphism

A többértékűség vagy többalakúság további eszközöket ad a kezünkbe. Ha például több osztályunk is van amikben hasonló munkát akarunk elvégezni, keresztelhetjük azonos névre is a metódusainkat, miközben a tevékenységet végrehajtó művelet **megvalósítása** az osztályokban különböző lehet. Így mindig azon múlik a végeredmény, hogy melyik osztály példánya triggerelte (hívta meg) a metódust.

Másik eset a felülírás ([overriding](#)), amikor az azonos ősoosztályhoz tartozó osztályokban egy tulajdonság értékét, vagy munkavégző egységét felüldefiniáljuk.

## Nos, akkor hasznosítsuk a tudást...

Az előbb taglalt háromfajta viselkedés okos kihasználása együtt azt eredményezi, hogy programkódunk sokkal struktúráltabbá, könnyebben bővíthetővé és jól karbantarthatóvá válik. Demonstrálandó az állítást kicsiben, rágjuk át magunkat a bevezető kódoknál kicsit bonyolultabb osztályhálón. Fontos, hogy még a fejesugrás előtt tisztázzuk: a kódot nem soronként kell olvasni, hanem egységenként és ide-oda ugorva attól függően hova vezet minket a kódrészlet, vagy egy kérdőjel a fejünkben. Az osztályháló fogalma nem véletlen: a szorosan összetartozó/együttműködő osztályok lefedik egy megvalósítandó feladat minden zegét-zugát és egyik sem létezhet a másik nélkül, kibogozhatatlanul összefonódnak egymással.

## Lássuk a medvét!

Az alábbi kódon végigkövetjük az eddigieket új információt csatolva a meglevők mellé, amikor szükséges. A kék nyilakkal ugorhatunk a vonatkozó részekre (ugrás után a lap tetején keresgélj és ne használd a böngésző pdf olvasóját, mert az nem ugrik pontosan). Az OOP szemléltetés szempontjából irreleváns kódrészek szürkék.

/A kód pszeudokód ismét, hogy minden járulékos, programnyelvre jellemző sallangtól mentes lehessen. Az osztályaink két képzeletbeli beépített osztályt használnak: a Graphic osztályt terjesztik ki, hogy rajzolni tudjunk bennük és a Point osztályt, hasonló az előzőekben már definiált osztályunkhoz. Nem kell minden egyes osztályt magunknak megírunk. Segítségünk lehet a nyelv, amit választunk, a különféle keretrendszerek (framework) és könyvtárak (library)./

package factory

```
import factory.HandleFactory;  
import display.Graphic;  
import geom.Point;
```

```
public class Main extends Graphic
```

```
private var factory:HandleFactory;  
private var handles:Array.<Graphic>;  
private static var _coordinates:Array.<Point> = [  
    new Point( 50, 20 ),  
    new Point( 50, 40 ),  
    new Point( 50, 60 ),  
    new Point( 50, 80 ),  
    new Point( 50, 100 ),  
    new Point( 50, 120 ) ];
```

```
public function Main()  
    factory = new HandleFactory();  
    handles = factory.createHandles();
```

```
static public function get coordinates():Array.<Point>
    return _coordinates;
```

/A statikus (static) metódusok nem a példányok részei, hanem az osztályé. Ez a gyakorlatban azt jelenti, hogy a példányon keresztül (példánynév.függvénynév) nem elérhető a metódus. Általában független munkák elvégzésére (számolások, speciális adatok készítése, stb.) használjuk őket, amik elérését nem érdemes egy példányhoz kötni, mert egyrészt magunkat akadályozzuk vele, másrészt nem egy példány feladata, hogy tool-ként működjön a program egészére nézve./

package factory

```
import factory.handle.HandleCreator;
import factory.handle.HandleCreatorCircle;
import factory.handle.HandleCreatorTriangle;
import display.Graphic;
import geom.Point;
```

```
public class HandleFactory extends Graphic
```

```
    private var _handles:Array.<Graphic>;
```

```
    public function HandleFactory()
        _handles = new Array.<Graphic>();
```

```
    public function createHandles():Array.<Graphic>
        var coordinates:Array.<Point> = Main.coordinates;
        var trianglecreator:HandleCreator;
        var circlecreator:HandleCreator;
        var triangle:Graphic;
        var circle:Graphic;
```

```
        for ( var i:int = 0; i < coordinates.length; i += 2 )
            trianglecreator = new HandleCreatorTriangle();
            circlecreator = new HandleCreatorCircle();
```

```
            triangle = trianglecreator.doStuff();
            circle = circlecreator.doStuff();
```

```
            triangle.x = coordinates[ i ].x;
            triangle.y = coordinates[ i ].y;
            triangle.rotation = i * 20;
            circle.x = coordinates[ i + 1 ].x;
            circle.y = coordinates[ i + 1 ].y;
```

```
            _handles.push( triangle, circle );
```

```
        return _handles;
```

/Egy új osztály új típust is jelent egyben a kódban. A típusok lehetnek beépítettek (az adott nyelv elérhető részei) és sajátok. Egy attribútum típusának megadásakor használhatjuk az őosztály típusát is, a gyerek saját típusát is, de bárhol megállhatunk végig a családfán ott, ahol minden állapot leíró adat és a rajtuk végezhető műveletek elegendőek a további munkánkhoz. A típusmegadás módja szintén programnyelv függő, ebben a kódban az attribútum neve mögé került kettősponttal. Használat előtt a típusokat hozzáférhetővé kell tenni az osztályban, erre

általában az `import` kulcsszót használjuk./

A fenti osztályban mi a `trianglecreator` attribútum típusa? ↗

Melyik `factoryMethod` fut le a `trianglecreator.doStuff()` híváskor? ↗

Mi lesz a `handle` változó típusa az előbbi `doStuff` függvényben? ↗

És ezek után melyik metódus fut le a `handle.drawHandle` hívásakor? ↗

```
package factory.handle
```

```
import display.Graphic;
import errors.IllegalOperationException;
import utils.getQualifiedClassName;
```

```
public abstract class HandleCreator extends Graphic ↗
```

```
    public abstract function HandleCreator()
```

```
    public function doStuff():Graphic
        var handle:IHandle = factoryMethod();
        handle.drawHandle();
        return handle as Graphic;
```

```
    protected abstract function factoryMethod():IHandle
```

/A `HandleCreator` egy absztrakt osztály, azaz nem mindent tulajdonságát kell definiálnunk. Segítségével a származtatott osztályokban rendelkezésünkre áll a `factoryMethod` metódus, amit osztályonként másképp implementálhatunk./

```
package factory.handle
```

```
public class HandleCreatorCircle extends HandleCreator
```

```
    public function HandleCreatorCircle()
```

```
    override protected function factoryMethod():IHandle ↗
        return new CircleHandle();
```

```
package factory.handle
```

```
public class HandleCreatorTriangle extends HandleCreator
```

```
    public function HandleCreatorTriangle()
```

```
    override protected function factoryMethod():IHandle ↗
        return new TriangleHandle();
```

```
package factory.handle
```

```
import display.Graphic;
```


```
public class CircleHandle extends Graphic implements IHandle ↗
```

```
    private var _bgcolor:uint;
    private var _bgalpha:Number;
    private var _currenttarget:Graphic;
```

```
    public function CircleHandle()
        _bgcolor = 0xFFAA2A;
```



```
_bgalpha = 0.2;
```

```
public function drawHandle():void   
    graphics.clear();  
    graphics.beginFill( _bgcolor, _bgalpha );  
    graphics.lineStyle( 1, _bgcolor );  
    graphics.drawCircle( 0, 0, 4 );  
    graphics.endFill();
```


```
package factory.handle
```

```
import display.Graphic;
```

```
public class TriangleHandle extends Graphic implements IHandle
```

```
    private var _bgcolor:uint;  
    private var _bgalpha:Number;  
    private var _vertices:Array.<Number>;  
    private var _currenttarget:Graphic;
```

```
    public function TriangleHandle()  
        _bgcolor = 0xFFAA2A;  
        _bgalpha = .2;  
        _vertices = new Array.<Number>();  
        _vertices.push( -5,-5, 5,-5, 0,5 );
```

```
    public function drawHandle():void   
        graphics.clear();  
        graphics.beginFill( _bgcolor, _bgalpha );  
        graphics.lineStyle( 1, _bgcolor );  
        graphics.drawTriangles( _vertices );  
        graphics.endFill();
```

```
package factory.handle
```

```
public interface IHandle   
    function drawHandle():void;
```

A kód [kimenete](#) és az eredeti, működő ActionScript forráskód ezen a [linken](#) található. A futtatáshoz a todo.txt fájlban találsz segítséget. Élő kódot azért csatoltam az íráshoz, mert a saját tapasztalatom szerint a próbálkozások során lesz igazán kristálytisza a működés mikéntje.

## Egyéb hasznos tudnivalók

### Interface

Az interfészek az OO nyelvek elengedhetetlenül fontos részei. Mivel a típusbiztonságot a fordítóprogram ellenőrzi, nem tudunk hamis típust megadni egy példánynak. Viszont nem minden esetben hozhatunk létre rokonságot osztályok között egy szükséges típusazonosság, vagy egyéb apróságok miatt. Tehát szükségünk lett egy

olyan osztálytípusra, amin keresztül elérhető attribútumok és metódusok is, elfogadja típusként a fordító, de nem szükséges kiterjeszteni, azaz az örökösének lenni.

A kívánságlistára egy interfész a válasz: két vagy több, egymásról semmit sem tudó osztály között biztosít egy közös felületet és **használható típusdefinícióként** is. A közös felületet a benne felsorolt (de nem kifejtett) metódusok és attribútumok jelentik, amiket minden felhasználó osztálynak meg kell valósítania.

A hiányzó implementáció miatt nem példányosítható.

/Ha egy osztály több interfészt is használ, akkor a felületek minden egyes tulajdonságát implementálnia kell./

## Abstract class

Egy absztrakt osztály egyetlen szerepe, hogy közös ősként felhasználhassuk, nem lehet belőle példányt létrehozni. Tartalmazhat definíció nélküli tulajdonságokat hasonlóan az interfészhez, de megadhat részleges, vagy akár teljes implementációt is. A származtatott osztályok feladata, hogy a hiányzó definíciókat pótolják, a részlegeseket pedig kiegészítsék. A meglevőket ugyanúgy felhasználhatják, ahogy egyébként teszik az öröklő osztályok.

A hiányos implementáció miatt nem példányosítható.

Az interfész és az absztrakt osztály között tehát lényeges tervezési különbség van, hiába hasonlítanak „külsőalakra”. Az egyik a nem rokon osztályok között biztosít kapcsolatot, a másik a rokonokat segíti a felhasználható tulajdonságokkal.

/Nem minden nyelv biztosítja a lehetőséget egy elvont osztály készítésére, ilyenkor trükközni kell a meglevő eszközök használatával – a forráskódban lesz rá példa./

## Meddig él egy példány?

Az osztályokból (halott struktúrákból) létrehozott ideig-óráig élő példányok élete a konkrét elpusztításukig tart. A memóriából soha, egyik nyelven sem ürülnek a példányra vonatkozó bejegyzések, amíg egyetlen hivatkozás is van rájuk és fut a program. Tehát nem elég kiadni a parancsot, /programnyelv specifikus, de általában példány = null, vagy delete példány/ hanem előtte el kell vágni az összes köteléket, ami a példányt az élettérhez köti, az összes általa használt „tárgyat” meg kell szüntetni, vagy parancsba adni a kötelék feloldását.

# Mikor érdemes OO-zni?

/A struktúrált program ellentéte (a közhiedelemmel ellentétben) nem az objektumorientált program, hanem a struktúrálatlan program, amelyik közvetlen vezérlésátadást (goto utasítás) tartalmaz./

A struktúrált / procedurális programozásnál előny, hogy könnyen átlátható a forráskód, mivel nagyjából egy helyen van minden rész összegyűjtve. Nagyobb projektekben azonban már kényelmetlenséget okozhat, hogy egy-egy algoritmust többször (redundancia), vagy többféleképpen is le kell írunk, holott lényegi változtatást nem tartalmaz az új. Függvényekkel, szubrutinokkal a problémát részben lehet orvosolni, de az igazi megoldást ilyen helyzetben az objektumok használata jelenti az OOP alaptulajdonsága, a polimorfizmus miatt. OO-gyanús helyzet még, ha túl sok „if” kell a program gyökerében a folyamat mederben tartásához, vagy várhatóan többször kérnek majd módosított verziókat a programból, illetve ha jópár ezer soros kódra számíthatunk és az ezzel együtt járó káoszra.

De a legjobb, ha nem menekülési útvonalként, hanem hasznos barátként tekintünk az OOP-re és előre számolunk a lehetőségeivel amikor a feladatkiírás alapján megéri alaposabban tervezni.

## ... és meddig lehet ezt még bonyolítani?

Tulajdonképpen az lenne a lényeg, hogy ne legyen túlbonyolítva. A mostani első hidegzuhany után letisztul majd a pár lényeges információra szűkített mondandó és az azt illusztráló kód. Az OOP-t szokni kell. Akár több évig is.

A rengeteg eszközből (design pattern, library, framework, stb.) is csak annyit érdemes használni, amennyit a feladat megoldása megkíván és egyszerűbbé teheti életünket.

Ezenkívül mindig érdemes kész megoldásokat és ötleteket lesni a netről, nem kell ragaszkodni a spanyolviasz feltalálásához.

author: jdk

Az írást hiánypótlásnak szántam, mivel sem anno, sem most nem találtam olyan segítséget az OOP megértéséhez, ami viszonylag egyszerű megközelítésben, röviden, de összesítve feltárja az összes lényeges jellemzőt. Pláne nem magyar nyelven, vagy programnyelvtől függetlenül. Nem vagyok benne biztos, hogy ez nekem sikerült, így minden javaslatot szívesen veszek egy jobb végtermék reményében :)