Milyen két - a jogosultságra vonatkozó - biztonsági megoldást támogat a .NET keretrendszer? Ismertesse őket röviden!

- **szerep alapú biztonság (role based security):**
    - a felhasználó a rendszerben betöltött szerepén és jogain alapuló biztonsági mechanizmus.
    - A felhasználókat csoportokba rendezik szerepek alapján.
    - A .NET (akárcsak a Windows) szál szinten kezeli.
    - Nem véd a rosszindulatú kódtól.
    - Nincsenek felkészítve a mobil kódokra.
    - Ellenőrzés lehet:
        - Dekleratív:
            - felhasználói szerepeket megkövetelhetjük
            - felhasználó szinten is akár
            - minden azonosított felhasználóra engedélyezhetünk
        - Imperatív:
            - ha nincs elegendő információnk fordítási időben a felhasználóról, akkor programkódból is ellenőrizhetjük a feltételeket.
- **kóderedet alapú biztonság (code access security):**
    - Képes meghatározni, hogy a kód és a meghívott kód mit tehet, valamint képes egyértelműen azonosítani a kódot, nemcsak a felhasználót.
    - Szerelvényekhez kell meghatározni, hogy milyen jogaik vannak.
        - Ezt a rendszergazda állítja be a biztonsági házirendben. (nem egyesével)
        - Ebben a kód eredete alapján adhatunk jogokat a szerelvényeknek.

.NET szerelvény (assembly) fogalma, szerepe, típusai, azonosítása

- általában egy .dll vagy egy .exe, de lehet több is
- IL kód
- metaadatok .NET osztályokból
- erőforrások (.jpg, .txt)

Minden alkalmazás szerelvényekből épül fel:

- egy névtér több szerelvényben is lehet
- egy szerelvényben több névtér is lehet
- hivatkozhat más szerelvényekre

Szerelvény fájlok:

- egy fájl
- memória fájl, pl. scriptelésnél
- több fájl: egy mappában a manifesttel, a biztonságosság miatt hash készül a hivatkozott állományokról

Szerelvény információk: a manifest

- név (általában kiterjesztés nélküli állománynév)
- verzió (major, minor, build number, revision)
- támogatott nyelv és kultúra
- processzor és OS

- szerelvény referenciák: név, verzió, nyilvános kulcs (ha megosztott), hash algoritmus azonosító (ha megosztott), szerelvényhez tartozó modulok listája, hash, egyéb (kiadó, leírás)

Verziókezelés:
- ha új verziót telepítünk, akkor is a régit használja
- a kibocsátó cég átirányíthat
- a rendszergazdák felüldefiniálhatják

1. **Privát szerelvény:**
   - egyetlen alkalmazás használja
   - a neve azonosítja
   - az alkalmazás mappáiban keresi (konfigurációs állományban más elérési útvonal is megadható)
   - egyszerűen telepíthető

2. **Azonosított szerelvény (megosztott, erős névvel ellátott)**
   - több alkalmazás használja --> "dll hell"
   - erős név teszi egyedivé: név, fejlesztő cég nyilvános kulcsa, verzió szám, opcionálisan nyelv és kultúra
   - digitális aláírás a fejlesztő cég privát kulcsával --> integritásvédelem
   - csak azonosított szerelvényekre hivatkozhat
   - %WINDIR%\Assembly mappában
   - a több alkalmazás által használt komponensek a %WINDIR%\System32 mappában

A típusok szerelvényekhez kötődnek, nem névterekhez. Ugyanabból a szerelvényből több futhat egymás mellett, akár egy folyamaton belül is. A fejlesztő minimális engedélyeket kérhet.

Felügyelt környezetek

a) **Ismertesse a fordítási és végrehajtási (futtatási) lépéseit a .NET környezetben. Ennek során adja meg mit értünk IL (Intermediate Language) kód alatt!**

Fordítás:
Forrás kód (.cs, .vb, …) → nyelvi fordító → Köztes kód (IL) + metaadatok (.dll, .exe) /szerelvény/

Végrehajtás:
Köztes kód (IL) + metaadatok (.dll, .exe) /szerelvény/ → JIT fordító → Natív kód

IL segítségével n+m (az n*m helyett) translatorra van szükség, hogy n nyelvet m platformra implementáljunk. IL növeli a kompaktságot is, mert kompaktabb lehet, mint az eredeti kód.
- processzor és architektúra független
- kiértékelő verem alapú
- ellenőrizhető
- továbbfordításra tervezték
- nyelvfüggetlen
- objektumorientáltság jellemzi
- metaadat: típusok leírása, tagváltozók, metódusok leírása
- könnyű visszafejteni (pl. reflector)

b) Adja meg a futtatókörnyezetek alkalmazásának három fontos előnyét!
c)

- **Hordozhatóság**
- **Kompaktság**
- **Hatékonyság**
- Biztonság
- Együttműködés
- Rugalmasság

d) Hogyan kezeli az OS a bill.leütést, hogy adja át a felügyelt környezetnek?
Az „R” billentyű lenyomása → alkalmazás üzenetsorába WM_KEYDOWN üzenet → a rejtett üzenetkezelő ciklus kiveszi a sorból → az ablak/form megkapja az üzenetet (WndProc) → virtual void OnKeyDown (KeyEventArgs e) hívása → event KeyEventHandler KeyDown elsütése

e) Írj egy Form alapú programot, ami MessageBox-ban megjeleníti a leütött billentyűt!

```
public partial class MainForm : Form
{
   public MainForm()
   {
      InitializeComponent();
      this.KeyDown += new KeyEventHandler(this.MainForm_KeyDown);
   }

   protected override void OnKeyDown(KeyEventArgs e)
   {
      base.OnKeyDown(e);
      MessageBox.Show("A billentyű (virt. fv.): " + e.KeyCode.ToString());
   }

   private void MainForm_KeyDown(object sender, KeyEventArgs e)
   {
      MessageBox.Show("A billentyű (eseménykez.): " + e.KeyCode.ToString());
   }
//...
}
```

f) Ismertesse röviden a Windows Forms saját vezérlők készítésének a lehetőségét!
g)
Három féle módszer van rá:

- **Control osztályból leszármaztatás:**
  - Akkor használjuk, ha egy teljesen új vezérlőelemet szeretnénk létrehozni.
  - Csak a minden Controlra közös tulajdonságokat kapjuk meg.
  - Adhatunk hozzá új eseményeket és tulajdonságokat.
  - A rajzolás is a mi feladatunk.
  - pl.: egy aktuális időt mutató címke
- **Adott vezérlőből leszármaztatás:**
  - Akkor használjuk, ha már létező vezérlőelemet szeretnénk testreszabni.
  - Csak a speciális viselkedést kell megvalósítani.
  - Adhatunk hozzá új eseményeket, tulajdonságokat.
  - pl.: egy speciális szövegablak, ami élénk háttérrel jelenik meg, ha érvénytelen email címet ír be a felhasználó

- **UserControl készítés:**
  - Akkor használjuk, ha újrafelhasználunk, vagy ha összetett felhasználói felület modularizálásának eszköze.
  - A vezérlőelem maga is egy űrlap, tartalmazhat vezérlőelemeket.
  - Tervezési időben vizuálisan elkészíthetjük összetett vezérlőelemeinket, pont úgy, ahogy egy formot is elkészítenénk.
  - A tartalmazott vezérlőelemet private láthatóságúak.
  - pl.: FilePicker vezérlő: tipikusan együtt előforduló vezérlőelemek összekötése.

Ismertesse a C# nyelv property fogalmát! Kódrészlettel illusztrálja a válaszát!

Property (tulajdonság) fogalma: ezek segítségével az osztályok tagváltozóihoz férhetünk hozzá szintaktikailag hasonló módon, mintha egy hagyományos tagváltozót érnénk el. Lehetőségünk van arra, hogy az egyszerű érték lekérdezés vagy beállítás helyett metódusszerűen implementáljuk a változó elérésének módját.

- mezőeléréshez funkció rendelése
- tagváltozó elrejtése, lekérdezése és manipulálása a tulajdonságon keresztül
- származtatott értékek
- hozzáférés szabályozása

```
string Name
{
    get { return name; }
    set { name = value; }
}
```

C#: indexer, boxing, attribútum + példakód mindre!

Indexer:

- konzisztens mód a containerek építésére
- a tulajdonságok ötleten alapszik
- indexelt elérést biztosít a tartalmazott objektumokhoz
- az index minősítő bármilyen típusú lehet

Példa:

```
object this [string index]
{
   get { return Dict.Item( index ); }
   set { Dict.add( index, value ); }
}
```

Dobozolás (boxing):

- érték szerinti típusok bedobozolhatóak és utána kidobozolhatóak
- ezzel lehetővé válik az érték típusú objektumok referencia szerinti átadása
- alapja az, hogy minden alaptípus objektum is
- dobozba dobjuk, tehát az értéket és "referenciáljuk"

Példa bedobozolásra:

```
object BoxedValue = value;
```

Példa kidobozolásra:
```
value = (double) BoxedValue;
```

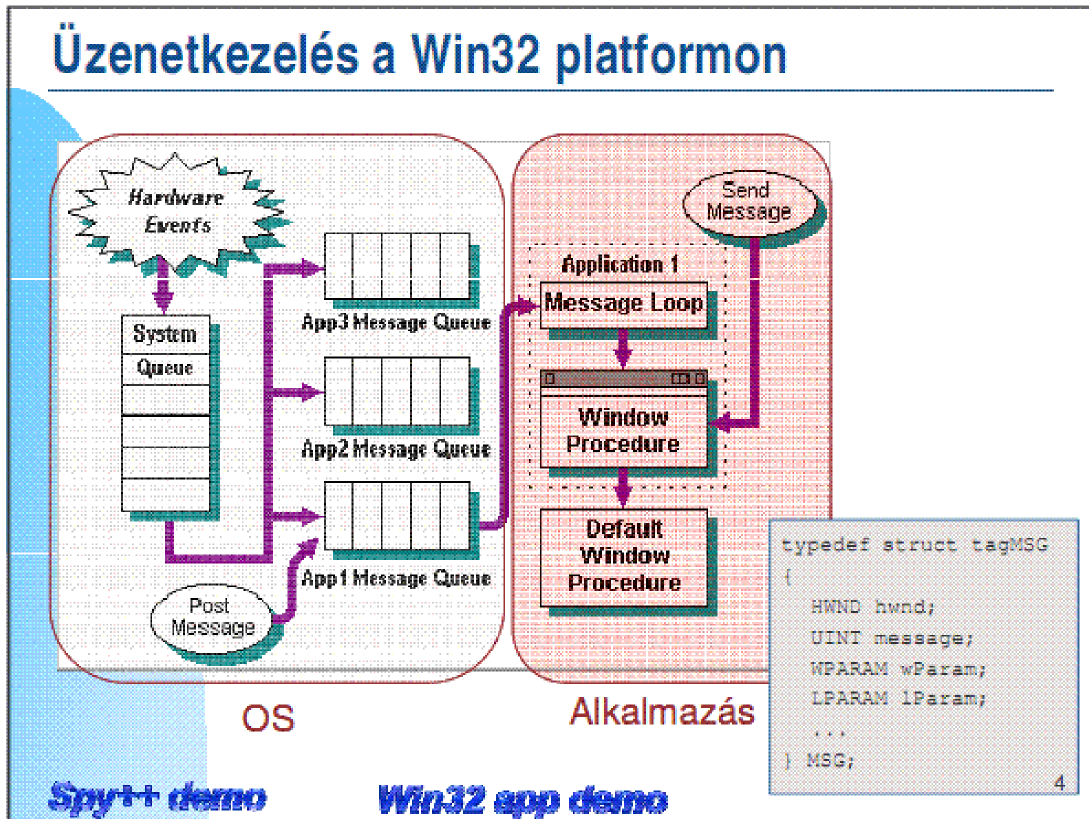
Attribútum:

- deklaratív jelleggel metaadatokat közölhetünk a kód bizonyos részeire vonatkozóan
- minden nyelvi elemhez rendelhető (típushoz, osztályhoz, interfészhez, metódushoz, stb.)
- megváltoztathatja a fordító viselkedését, VAGY futási időben lekérdezhető
- saját attribútumok alkothatók (ilyenkor kötelezően le kell kérdezni)
- általában az attribútumokkal csak a CLR számára szeretnénk információkat közölni
- felhasználás: natív kóddal való együttműködés testreszabása, perzisztens osztályok létrehozása, metódusszintű jogosultság ellenőrzés, tranzakcióban részt vevő osztályok megjelölése, web szolgáltatások

Példa (sorosíthatóság):
```
[Serializable]
int a;
```

Példa (szerepkör ellenőrzés):
```
[PrincipalPermission(SecurityAction.Demand, Role="Admin")]
public void DeleteUser() { /* ... */ }
```

Win32 natív üzenetkezelés + üzenet, üzenetsor, ablakkezelő függvény definíciója



Üzenet:
Eseménykor generálja a Windows. Tartalma:

- típus
- keletkezési idő
- melyik ablaknak szól az üzenet.

A billentyűzet és egér üzenetei mindig aszinkron módon dolgozódnak fel. Ezeket az eseményeket az üzenetté való konvertálása után a rendszer egy rendszer várakozási sorba (system queue) helyezi el. A system queue-ba érkező üzeneteket az operációs rendszer továbbítja az applikációk szálainak üzenet soraiba.

Üzenettípusok:

a) **Queued:** az üzenet bekerül az üzenetsorba és onnan továbbítódik az ablakkezelő függvénynek --> üzenet feladás, "post a message".
b) **Not-queued:** az üzenetsor kikerülésével közvetlenül az ablakkezelő függvény kapja meg az üzenetet --> üzenet küldés, "send a message".

Üzenetsor:
Minden szál rendelkezik egy FIFO üzenet sorral, amibe bekerülnek a szálhoz érkező üzenetek. Innen továbbítódnak a megfelelő ablakkezelő függvényhez. Ezeknek az üzeneteknek a feldolgozása aszinkron módon történik.

Üzenetkezelő ciklus:
Minden applikáció (minden GUI szál) tartalmaz egy üzenetkezelő ciklust, ami kiszedi az üzeneteket az üzenetsorából, és továbbítja őket a megfelelő ablakkezelő függvénynek.

Alapértelmezett ablakkezelő függvény:
Olyan események esetén hívjuk meg, melyek érdektelenek az alkalmazás szempontjából, és így az alapértelmezett módon szeretnénk lekezelni, pl. : ablak átméretezés, minimalizálás, menük megjelenítése, stb…

Callback függvény:
Kitüntetett szerepű függvény. Az operációs rendszer bejegyzi magának, esemény bekövetkeztekor pedig meghívja. Az eseménykor keletkezett üzenetből lehet tudni, hogy ki kapja az eseményt, és mik a paraméterek.

Egy egyszerű statikus struktúrából csinálj példakódot C++/C#/Java nyelven!

```
struct Point{
   double X;
   double Y;
   void MoveBy(double dX, double dY){
      X+=dX; Y+=dY;
   }
}
public static Point p;
p.X = 1.2f;
p.Y = 3.4f;
```

Egy egyszerű generikus struktúrából csinálj példakódot C++/C#/Java nyelven!

```
public struct Point<T>{
   public T X;
   public T Y;
}
public static Point<float> point;
p.X = 1.2f;
_p.Y = 3.4f;
```

Eseményvezérelt programozás és grafikus megjelenítés

a) **Érvénytelen terület fogalma:** korábban takarásban lévő, láthatóvá vált ablakrészek. Mivel a memória korlátok miatt, az OS nem jegyzi meg az ablakok rajzolatát, így az érvénytelen területeket újra kell rajzolni. Amikor egy érvénytelen terület keletkezik, akkor WM_PAINT üzenetet kap az ablak (natív Win32), ami a Paint eseménynek felel meg. *Ezt használjuk ki akkor is, ha újra akarunk valamit rajzoltatni. Hívunk egy Invalidate() függvényt, ami érvényteleníti a területet.*

b) **Írjon olyan C# nyelvű alkalmazásrészletet, amely a (10,20) koordinátában megjelenít egy közepesen szürke színnel kitöltött 10 pixel oldalhosszúságú négyzetet. A négyzet színe minden „r” billentyű megnyomására legyen egyre sötétebb. A megjelenítés GDI-re épüljön!**

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    protected override void OnKeyDown(KeyEventArgs e)
    {
        base.OnKeyDown(e);
        if(e.KeyCode == Keys.R) {
            if(if == 0) i = 200;
            if -= 10;
            Invalidate();
        }
    }
    private Brush brush1;
    private int i = 200;
    protected override void OnPaint(PaintEventArgs e)
    {
        base.OnPaint(e);
        using (brush1 = new SolidBrush(ConsoleColor.FromArgb(i, i, i)))
        {
            e.Graphics.FillRectrangle(brush1, 10, 20, 20, 30);
        }
    }
}
```

c) **Írjon olyan C# nyelvű alkalmazásrészletet, amely a (10,20) koordinátában megjelenít egy piros színnel kitöltött 50 pixel oldalhosszúságú négyzetet. A négyzet színe minden egérkattintáskor legyen egyre sötétebb piros (végül fekete). A megjelenítés a GDI-re épüljön (nem használhatja a Label, TextBox, stb. vezérlőket)! Csak a megoldáshoz szorosan hozzátartozó kódrészletet adja meg!**

```
public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        protected override void OnMouseDown(MouseEventArgs e)
        {
            base.OnMouseDown(e);
            if (i>0) i -= 5;
            Invalidate();
        }

        private Brush brush1;
        private int i = 255;

        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);

            using( brush1 = new SolidBrush(Color.FromArgb(i, 0, 0) ) )
            {
                e.Graphics.FillRectangle(brush1, 10, 20, 50, 50);
            }
        }
    }
```

d) Írjon olyan C# nyelvű alkalmazásrészletet, ami a (20, 20) koordinátában megjeleníti, hogy a legutóbbi egérkattintás óta hány másodperc telt el! A megjelenítés GDI-re épüljön (nem használhatja a Label, Textbox, stb. vezérlőket). Csak a megoldáshoz sorosan kapcsolódó kódrészeket adja meg!

```
DateTime lastClick;
        String strDeltaTime;
        private void Form1_Load(object sender, EventArgs e)
        {
            lastClick = DateTime.Now;
            strDeltaTime = "0";
        }

        private void Form1_MouseClick(object sender, MouseEventArgs e)
        {

            TimeSpan deltaTime = DateTime.Now.Subtract(lastClick);
            lastClick = DateTime.Now;
            strDeltaTime = deltaTime.Seconds.ToString();
            Invalidate(); // érvényteleníteni kell az ablak területet, hogy az új érték látszódjon
(lényegében ezzel "hívjuk" meg az OnPaint metódust)
        }

        protected override void OnPaint(PaintEventArgs e) {
            e.Graphics.DrawString(strDeltaTime, this.Font, new SolidBrush(Color.Black), 20, 20);
            base.OnPaint(e);
        }
```

Szálkezelés

a) Adja meg a szálbiztos osztály fogalmát egy-két mondatban!

Szálbiztos: megosztott közös erőforrásokhoz (pl: memória, fájl) egyidőben csak egy szál férhessen hozzá, hogy a konzisztencia ne sérüljön (sosem tudhatjuk, hogy mikor veszi el az ütemező a száltól a futás jogát)

Szálbiztos osztály: olyan osztály, ami úgy lett megírva, hogy többszálú környezetben is garantálja a helyes működést, biztonságosan használható. Garantálja a konzisztenciát és felhasználás során már nem kell zárolni.

b) Szálbiztos-e az alábbi C# nyelven írt osztály? Válaszát röviden indokolja!

```
class ThreadSafeClass
{
    static long x = 0;
    object SyncObject = new Object();
    public void IncrementX()
    {
        lock (SyncObject) { x++; }
    }
}
```

c) Példa szál indítására

A parancssori argumentumok számától függően paraméteres, illetve paraméter nélküli szálat indítunk el:

```
class ThreadTestClass
{
    public static void Main(string[] args)
    {
        Thread t = null;
        if (args.Length == 0)
        {
            t = new Thread(new ThreadStart(ThreadMethod1));
            t.Start();
        }
        else
        {
            t = new Thread(new ParameterizedThreadStart(ThreadMethod2));
            t.Start(args[0]);
        }
    }
    public static void ThreadMethod1()
    {
        Console.WriteLine("Thread without parameter.");
    }

    public static void ThreadMethod2(object param)
    {
        Console.WriteLine("Thread with parameter: {0}", param.ToString());
    }
}
```

d) Írjon programot, ami egy háttérszálban egy perc alatt elszámol 1-től 60-ig és az aktuális értéket kiírja konzolra.

```
public class Program
{
  public void static Main(string[] args)
  {
       Thread t=new Thread(Szamol);
       t.Start();
       t.IsBackground=true;
  }
  public void static Szamol()
  {
       private static int szam=0;
       if(szam<60)
    {
```

```
            Sleep(1000);
            szam++;
            Console.WriteLine("A számláló értéke: {0}",szam.ToString());
          }
     }
}
```

e) **Egészítse ki a programot úgy, hogy egy tetszőleges billentyű leütésével a számolás megszakítható legyen, de a program ne lépjen ki, csak újabb billentyű lenyomásakor (billentyű leütésre várakozni a ReadKey() hívással lehet).**

A main függvényben a `t.IsBackground=true;` után:

```
ReadKey();
t.Exit();
t.Join();
Console.WriteLine("A számolás megszakadt.");
ReadKey();
```

Ismertesse a kölcsönös kizárás, kritikus szakasz, Lock, Mutex, Semaphore zárolási technikákat, és ismertesse a ReaderWriterLock osztályt egy-két mondatban!

Kölcsönös kizárás: egy közös erőforráshoz egyidőben csak annyi magában szekvenciális kód férhessen hozzá, amennyivel az erőforrás működése még helyes marad.

Kritikus szakasz: az a kódrészlet, amelyből a megosztott erőforráshoz hozzáférünk, amelyre garantálni kell az atomi/oszthatalan hozzáférést.

Lock:

- Egy objektum paramétert kell neki adni, ami csak referencia típus lehet.
- Megvizsgálja, hogy zárolva van-e az objektum. Ha nincsen, akkor zárolja és tovább fut, ha zárolva volt, akkor vár.
- A várakozás nem használ CPU időt.
- A várakozó szálak egy sorba kerülnek, „érkezési sorrendben" kapnak hozzáférési jogot.
- A lock blokkból való kilépéskor oldja a zárolást.
- Zároló szál ismételten hívhat lock-ot.
- Védeni kell a tagváltozókat:
  - Statikus változókat statikus tagváltozóval – osztályszintű zár
  - Nem statikusokat nem statikus tagváltozóval – objektumszintű zár
- Más szálnak jelzésre nem alkalmas.

Mutex:

- WaitHandle leszármazott
- Olyan mint a lock, csak eltérő folyamatok szálai között is működik.
- Lock-nál sokkal lassabb.
- A WaitOne()-nal lehet lefoglalni (ami atomi módon zárolja, illetve blokkol, ha foglalt volt), elengedni a ReleaseMutex()-szel lehet.

Semaphore:

- WaitHandle leszármazott.
- Olyan mint a mutex, de egynél több (N darab, megadható) hozzáférést engedélyez, ha maximum N darab hozzáférést szeretnénk biztosítani.

- A WaitOne()-nal lehet rá várakozni. Ha még szabad a szemafor, nem blokkol, eggyel csökkenti a szemafor számlálót. Ha nem szabad, akkor blokkol. Elengedni a ReleaseSemaphore()-ral lehet (ez növeli a számláló értékét eggyel).

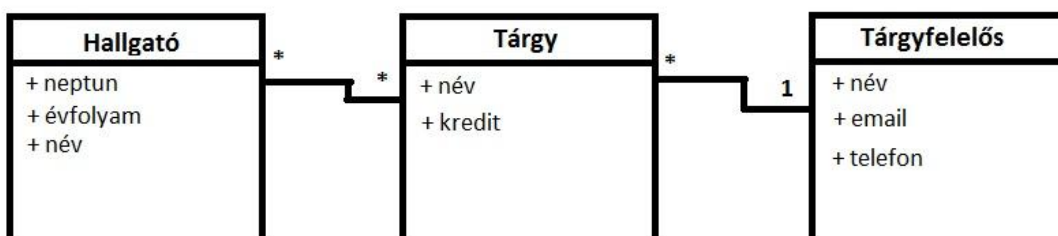
ReaderWriterLock osztály:

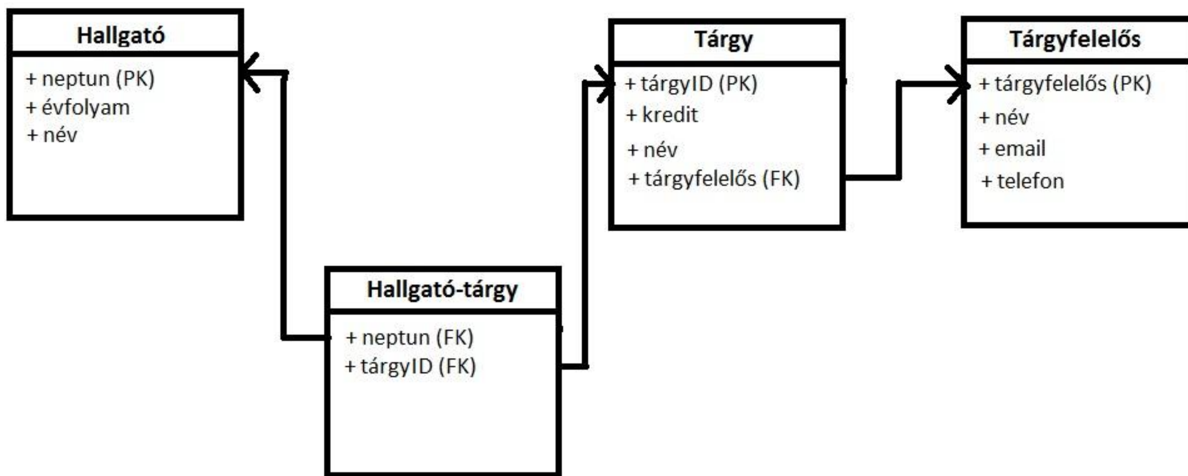
- Sok olvasóra optimalizást megoldás.
- Egyszerre több olvasó is hozzáférhet az erőforráshoz, de íróból csak egy, illetve az író kizárja az olvasókat is.
- pl.: ritkán módosított CACHE megvalósítására alkalmas.

Név	Cél	Folyamatok között is?	Sebesség
lock C# utasítás (Monitor.Enter/Monitor.Leave)	Biztosítja, hogy egy adott erőforráshoz/kódrészlethez egyidőben csak egy szál férhet hozzá.	nem	gyors
Mutex	Mint a lock, de folyamatok között is. Pl. annak megoldására, hogy egy alkalmazásból csak egy példány indulhasson.	igen	közepes
Semaphore	Mint a Mutex, de nem egy, hanem max. N hozzáférést engedélyez.	igen	közepes
ReaderWriterLock	Sok olvasóra optimalizált megoldás. Egyszerre több olvasó is hozzáférhet az erőforráshoz, de íróból csak egy (illetve az író kizárja az olvasókat is). Pl. ritkán módosított cache megvalósítása.	nem	közepes

Példán keresztül mutassa be az objektum relációs leképezést, adjon meg egy osztálydiagramot, amely tartalmaz 1-több, több-több kapcsolatot! Képezze le ezeket az adatbázistáblába!

1-több: a többes oldal táblájába felveszünk egy idegen kulcsot az egyes oldal táblájának elsődleges kulcsára.
Több-több: felveszünk egy új kapcsolótáblát, amely tartalmaz 1-1 idegen kulcsot az összekapcsolt táblák elsődleges kulcsára.





Tervezési minták

a) Adja meg röviden, hogy miben és hogyan segítenek a tervezési minták a tervezés során!

Tervezési minták: gyakran előforduló problémákat írnak le; annak környezetét és a megoldás magját, amit alkalmazva számos gyakorlati eset hatékonyan megoldható. A fejlesztés során a tervezés fázisában segítenek, méghozzá ezekben:

1. Megfelelő objektumokat megtalálni, definiálni
   Objektumok számának és méretének meghatározása
   Objektum interfészek definiálása,
   Objektum implementálása
2. Újrafelhasználás
3. Változtathatóság, kiterjeszthetőség

b) Jellemezze a Singleton tervezési mintát! Milyen általános problémákat old meg? Hogyan lehet implementálni C++, C# vagy Java nyelven?

Singleton tervezési minta: célja, hogy egy osztályból csak egy példányt lehessen létrehozni, és ehhez az egy példányhoz globális hozzáférést biztosítson.
Az Instance osztály-művelet (statikus!) meghívásával lehet példányt létrehozni, illetve az “egyetlen” példányt elérni. A Singleton::Instance().

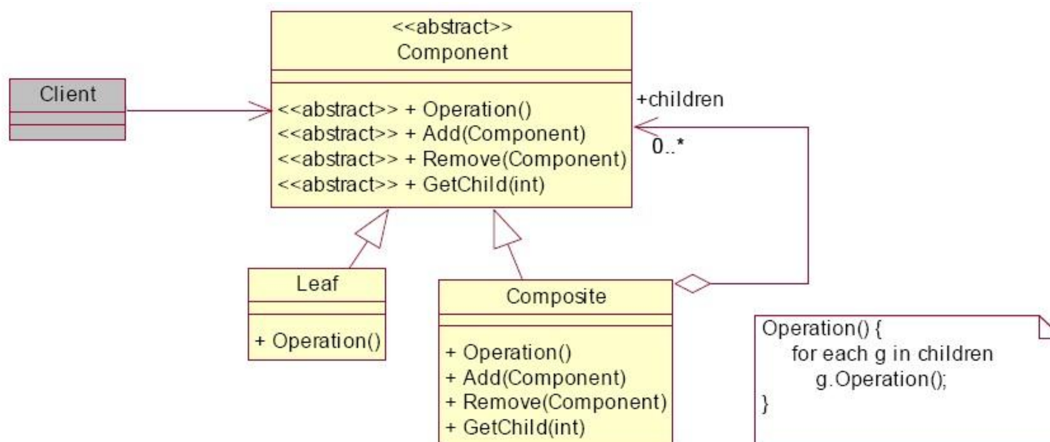
- mindig ugyanazt az objektumot adja vissza
- C++ esetén: Singleton()::Instance()
- C# esetén propertyvel célszerű: Singleton.Instance
- Java esetén: Singleton.GetInstance()

```
public class Singleton
{
    private static Singleton instance = null;
    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            instance = new Singleton();
            return instance;
        }
    }
    protected Singleton() { }
    public void Print() { }
}
```

```
//Használata:
Singleton s1 = Singleton.Instance;
//…
Singleton.Instance.Print();
```

c) **Jellemezze a "Composite" tervezési mintát! Mire ad megoldást a "Composite" tervezési minta? Mutassa be konkrétan vagy egy példán keresztül a minta működését! Ezen felül rajzolja fel a minta osztálydiagramját, valamint adja meg a mintában szereplő osztályok szerepét!**

- Rész-egész viszonyban álló objektumokat fastruktúrába rendezi
- A kliensek számára elérhetővé teszi, hogy az egyszerű és kompozit objektumokat egységesen kezelje.
- Akkor használjuk, ha a kliensek számára el akarjuk rejteni, hogy egy objektum egyedi vagy kompozit objektum-e
- Előnye, hogy elemi és összetett objektumokat egységesen lehet vele kezelni
- Például: egy olyan grafikai alkalmazás, amely lehetővé teszi összetett grafikus objektumok létrehozását.
    - Van egy absztrakt ősosztályunk, aminek a leszármazottjai lehetnek egyszerű és összetett objektumok is. Ha ki akarjuk rajzolni őket, akkor elég az absztrakt ősosztály rajzol függvényét meghívni, így minden objektum ki tudja rajzolni saját magát és a felhasználónak nem kell tudnia, hogy az adott objektum egyszerű vagy összetett volt-e.



d) **Jellemezze az "Adapter" tervezési mintát! Mire ad megoldást? Mutassa be konkrétan, vagy példán keresztül! Ezen belül rajzolja fel az osztdiagramját, és adja meg az osztályok szerepét!**

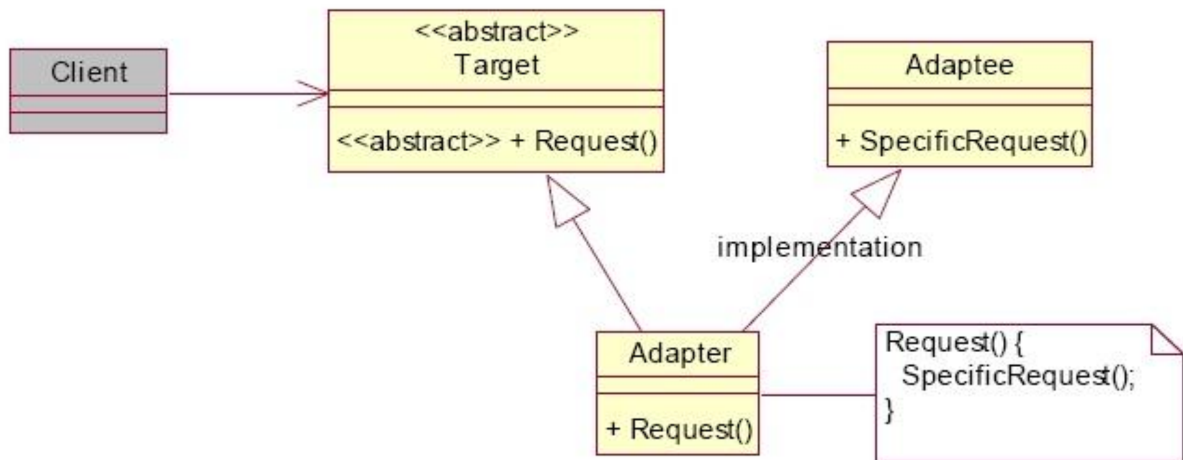
- Egy osztály interfészét olyan interfésszé konvertálja, amit a kliens vár. Lehetővé teszi olyan osztályok együttműködését, melyek egyébként az inkompatibilis interfészeik miatt nem tudnának együttműködni.
- Létezik Class és Object adapter is
- Példa: grafikus editor
    - Grafikus alakzatok (vonalak, poligon, szöveg) – a Shape osztályból származnak (vonal – LineShape, poligon – PoligonShape, szöveg – TextShape, stb)
    - TextShape megírása nehéz, viszont tegyük fel, hogy a framework egy sokoldalú TextView osztály, ami mindazt, tudja, amit a TextShape-től elvárunk.
    - Probléma: a TextView osztályt nem tudjuk közvetlenül felhasználni, mert nem megfelelő az interfésze, ugyanis nem a Shape osztályból származik (nem támogatja a Shape interfészt, emiatt nem tudjuk a többi Shape-el együtt egységesen kezelni).

- Megoldás: Adapter minta használata

- Class adapter példa
- Többszörös örökléssel oldja meg az adaptálást

<u>Szereplők:</u>

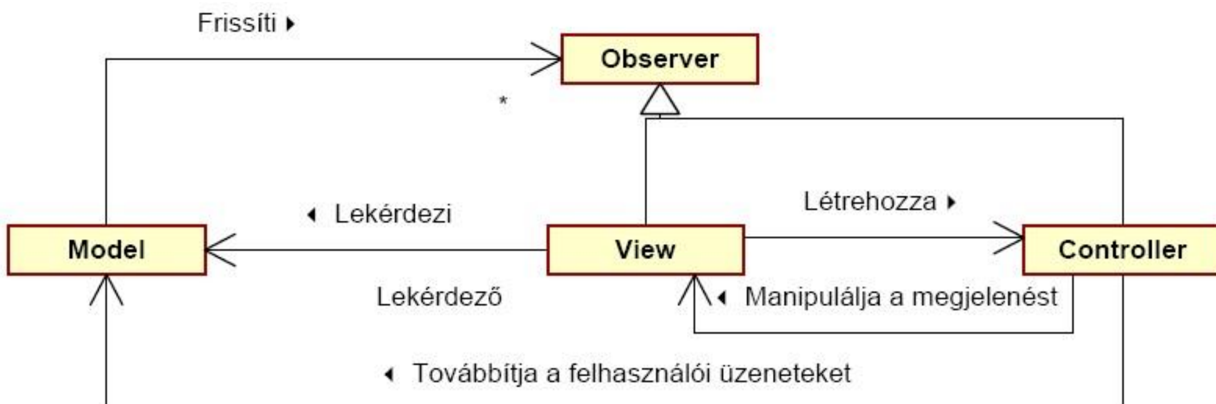
- *Adaptee (TextView):* interfész, amit adaptálni (illeszteni) kell
- *Adapter (TextShape):* illesztő, az Adaptee interfészt a Target interfésszé konvertálja
- *Target (Shape):* interfész, amit a kliens használ



Ismertesse az MVC (Model-View-Control) vagy a Dokumentum/Nézet (Document-View) architektúrát. Ennek keretében rajzolja fel a minta osztály és szekvenciadiagrammját. Adja meg az egyes osztályok szerepét!

<u>MVC (Model-View-Control):</u>

- felhasználói felülettel rendelkező alkalmazások
- alapigazság: ne drótozzuk bele GUI-ba az alkalmazáslogikát
- Model - alkalmazáslogika
- View – megjelenítés
- Contoller – interakció – kommunikáció a felhasználóval



Model: tartalmazza az adatokat (tagváltozók formájában), valamint olyan műveleteket, melyek ezeken az adatokon dolgoznak (pl. save, clear …)

View: Megjeleníti az adatokat, amiket a Model-ből olvas ki, nem tárolja ő is. Tartalmazhat olyan adatot, ami csak a nézetre vonatkozik.

Controller: A felhasználói interakciókat kezeli. Itt vannak a billentyűzet-, egér- és menüeseményeket kezelő függvények, melyek tipikusan a Model-be hívnak bele, vagy esetleg a View-ba (ha az adott esemény a View-ra vonatkozik, pl Zoom állítása).

Observer: Ha egy Controller megváltoztatja a Model-t, akkor a Model valamennyi View-t és Controller-t értesít, hogy frissítsék magukat a Model-ből. Ez garantálja, hogy minden View konzisztens nézetét mutatja az adatoknak, és a Contoller-ek is tudják tiltani/engedélyezni a felhasználói parancsok futtatását. A Model egy közös Observer típusú listában tárolja a View-okat és Contoller-eket.

- A Controller kezeli az eseményeket
- A Controller értesíti az eseményről a Modelt
- A Model értesíti megváltozásáról a Viewt/Viewkat
- A View(k) lekérdezi(k) a Model állapotát, majd megjeleníti(k) azt

Előnyök:

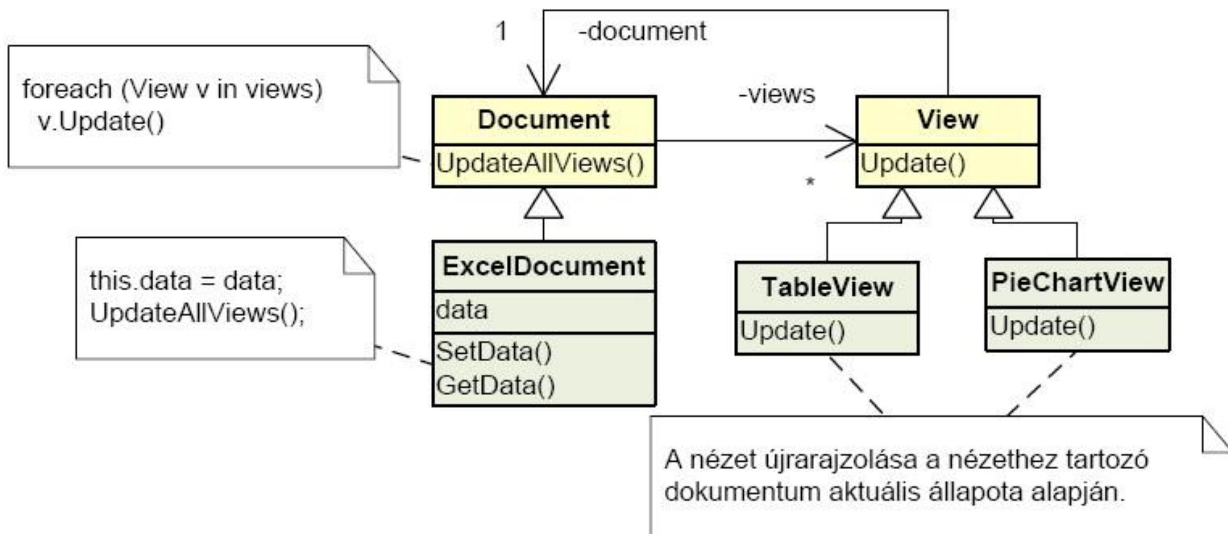
- Többféle nézete ugyanannak az adatnak
- Szinkronizált nézetek
- Függetlenül cserélhető, kibővíthető View-k és Controller-k
- A modell könnyebben tesztelhető
- Lehetséges framework

Hátrányok:

- Komplexitás növekszik
- Túl sok szükségtelen frissítés lehet
- View és Controller gyakran nem különválasztható, nem is célszerű. Megoldás a Controller és View összevonása: Document-View arcitektúra (MFC)

<u>Document-View architektúra (MFC):</u>

- **Document (Dokumentum):**
    - Feladata: az adatok tárolása, menedzselése. Modellnek is szokás nevezni.
    - Olyan osztály(ok), melyek az adatokat tagváltozóikban tárolják és olyan tagfüggvényekkel rendelkeznek, melyek kezelik ezeket az adatokat (pl. Load, Save), és elérhetővé teszik más osztályok számára. (pl. View számára)
- **View (Nézet):**
    - Feladata: az adatok megjelenítése a dokumentum adatai alapján és a felhasználói interakciók kezelése (pl. menük, egér, billentyűzet). A felhasználói interakciók során általában a Dokumentum tartalmát módosítja.
    - A View általában ablakként vagy tabfülként jelenik meg a kliensalkalmazásokban.



Document: Az osztály objektumai reprezentálnak egy-egy dokumentumot, és a views nevű listájában tárolja a beregisztrált nézeteket.

ExcelDocument:

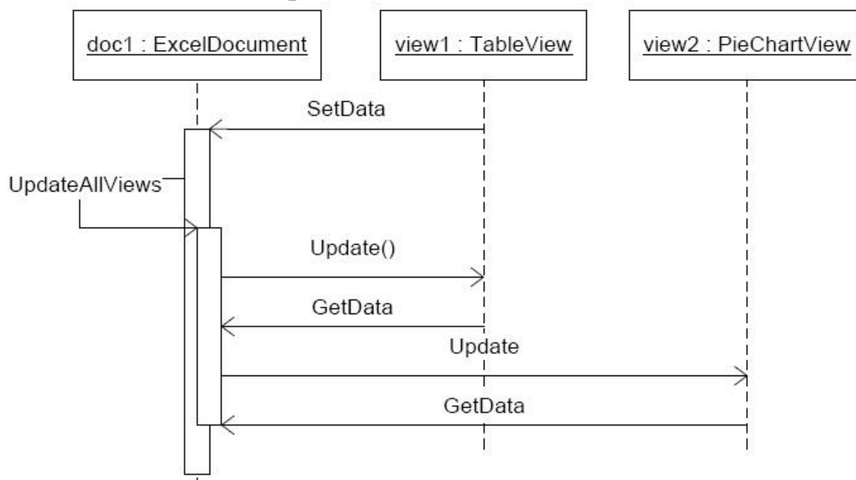
- Egy példa Document leszármazottra
- Tárolja a data, stb… tagváltozójában a dokumentum adatait (pl. cella értéke)
- Publikus lekérdező függvényeket biztosít a többi osztály számára az adatokhoz (pl. GetData)
- Publikus adatmódosító függvényeket biztosít a többi osztály számára (pl. SetData). Ezek módosítják a tagváltozókat, majd az UpdateAllViews() függvény meghívásával értesítik a többi nézetet a változásról.
- Az UpdateAllViews() frissíti a beregisztrált nézeteket

View:

- Az egyes nézetek közös őse vagy interfésze, lehetővé teszi egységes kezelésüket.
- Tartalmaz egy referenciát a dokumentumra, amin keresztül a leszármazott nézetek elérhetik a dokumentumot, melynek adatait megjelenítik.

TableView, PieChartView :

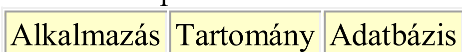
- A dokumentum teljes nézeteit reprezentálják.
- A View-ból származnak.
- Felüldefiniálják vagy implementálják a View Update() műveletét. Ebben a dokumentumban az aktuális állapota szerint frissítik a nézetet.



Ismertesse a vállalati információs rendszerek háromrétegű és kétrétegű architektúráját, melynek során adja meg röviden az egyes rétegek szerepét is!

<u>Háromrétegű architektúra</u>

- Külső séma - *alkalmazás*.
- *Alkalmazáslogikai alréteg*: opcionális. Ez egy interfész (homlokzat) a tartománymodellhez. Az interfész illeszkedik a GUI vezérlőelemeihez. A tartomány típusairól logikai típussá konvertál.
- Koncepcionális séma - *tartomány* (üzleti logika).
- Belső séma - *adatbázis*.
- Felépítés:



Alkalmazás	Tartomány	Adatbázis

<u>Előnyök:</u>

- Az alkalmazás kevésbé függ a fizikai adatszerkezettől.
- A referenciális integritás alkalmazásfüggetlenül kezelhető.
- Az adatbázis átszervezhető az alkalmazástól függetlenül.
- Tranzakciókezelés a DBMS feladata – ne nekünk kelljen implementálni.
- Ipari támogatottság (Microsoft, Sun).

<u>Hátrányok:</u>

- Kevesen használják.
- Nagyobb erőforrásigény.
- Többletmunka.
- Az objektumorientált adatbázisok jó felhasználási területe, de ezek teljesítőképessége kétséges.

<u>Kétrétegű architektúra</u>

- Megosztott adatbázis (file-ok, DBMS)
- Alkalmazások (hagyományos vagy 4GL nyelven)
- Felépítés:

Alkalmazás	Megosztott adatbázis

<u>Előnyök:</u>

- Az adatkarbantartás elkülöníthető az alkalmazásoktól.
- A már meglévő adatokat több szempontból (nézetből) is megjeleníthetjük.

<u>Hátrányok:</u>

- Az adatok integritása jól csak az adatbázis szintjén megoldható – ha nincs lehetőség tárolt eljárásokra, akkor ez problematikussá válhat.
- Ha az adatintegritás kezelését „bedrótozzuk” az alkalmazásba, nem lehet megváltoztatni a régi alkalmazásokkal való kompatibilitás miatt.
- Optimalizálás denormalizálással.
- Az alkalmazás az adatbázis fizikai felépítését is figyelembe kell vegye, pedig neki csak a szemantikát kellene.

Ismertesse a dinamikus webalkalmazások jellemzőit (definíció, architektúra, kilens oldal, szerver oldal)!

Dinamikus webalkalmazás:

- Feldolgozza a beérkező http kérést, és ennek részeként az esetleges felhasználói inputot, majd előállítja a kimenő HTML oldalt
- Szerver oldali logikát tartalmaz
- Az oldalakon található információk egy része gyakran adatbázisban tárolódik
- Kliens oldali logikát tartalmazhat (pl: JavaScript)

Kliens oldal:

- Tipikusan valamilyen böngésző fut -> HTML kódot, HTTP választ tud feldolgozni
  - Valamilyen szinten implementálja a HTTP szabványt
    - Kommunikálhat más protokollon is a szerverrel
    - Kompatibilitás és tűzfal barátság érdekében
  - Lehet RSS olvasó, letöltő, feltöltő, irodai alkalmazás, webszolgáltatás kliens
  - Lehet oldalba ágyazott kontroll (Flash, Silverlight)
- Letölti a kiegészítő fájlokat (kép, stíluslap, szkript stb.)
- Futtatja a kliens oldali kódot
  - Browser sandboxban
- A felhasználó akciói visszakerülnek a szerverre: postback vagy roundtrip
  - Pl. gomb megnyomására HTTP POST vagy URL paraméterek formájában

Szerver oldal:

- Feldolgozza a beérkező HTTP kérést -> input
  - Felhasználó által megadott input (URL, query string, form data)
  - Korábbi műveletek (session, cookie)
  - Kliens paraméterei (pl. User-Agent, Accept-Language header)
- Valamilyen nyelvű kód fut, aminek HTTP választ kell előállítania -> output
- Szerver oldali vagy külső erőforrásokat (adatbázis, web) használhat
- Kliens oldali erőforrásokat nem érhet el (browser sandbox, biztonság!)

Állapotkezelést (session ID) valahol meg kell oldani (HTTP nem támogatja):

- Valamilyen egyedi azonosító alapján
- Kliens oldalon
  - URL paraméter
  - Rejtett mező
  - Cookie
- Szerver oldalon
  - In-process a webszerver memóriájában
  - Külön folyamatban
  - Adatbázisban
- ASP.NET: web.configban tag

Webes alkalmazások

a) Milyen előnyei és hátrányai vannak a webes alkalmazásoknak? Írj 5 előnyt és 5 hátrányt!

Előnyök:

- egyszerű telepítés
- frissítés és javítás
- nyitottság
- szabványos távoli kommunikáció
- olcsóbb az üzemeltetés és hosszú távon a fejlesztés is
- mobil és nem mobil oldali kliensek egyaránt
- kliens oldali platformfüggetlenség
- távoli adminisztráció
- szabványos autentikáció használható

Hátrányok:

- kritikus válaszidő
- grafika erős támogatása (pl:GDI+)
- http és html korlátozottsága
- a védelem kritikus
- lokális erőforrások kezelése
- felhasználói hozzáállás
- infrastruktúra
- privát adatok hozzáférése
- adatok kezelése

b) Ismertesse a webalkalmazások állapotkezelését!

Szerver oldalon: Application objektum, Session objektum, adatbázis, fájl
Kliens oldalon: view state, rejtett mező, cookie, URL paraméterek

c) Hogyan applikálhatók a Session-ök állapotkezelésre?

- A kiszolgáló tárolja, felhasználónként elkülönítve
- Komplex objektumok tárolhatók benne
- Sok erőforrást (memóriát) igényelhet a kiszolgáló oldalon: skálázhatóság
- Nem lehet megosztani a felhasználók között
- Amennyiben nincs feltétlen rá szükség, érdemes kerülni a használatát
-

d) ASP.NET kiszolgáló oldali vezérlők (jellemzők, szerepük, működésük, példakód)

A felületelemek megjelenítéséért és viselkedéséért felelnek. Feldolgozzák a kliens oldalról érkező adatokat (pl. gomb nyomás esemény, elérhetővé teszik a böngésző által küldött felhasználói adatokat). Generálják a kliensnek küldendő HTML kódot (minden HTML kódra képződik le).

pl. TextBox létrehozása:

```
<asp:TextBox ID="tID" runat="server"></asp:TextBox>
```

- `tID` névvel lehet elérni a mögöttes kódból (.NET objektum)
- aspx fájlból HTML kódok segítségével testre-szabható a megjelenésük

Működés:

- első lekéréskor lefordul az aspx és a code-behind fájl egy temp könyvtárba DLL formájában.
- a lefordított kód feldolgozza a bejövő kérést és előállítja a választ.
- későbbi kéréseket a már lefordított kód szolgálja ki.