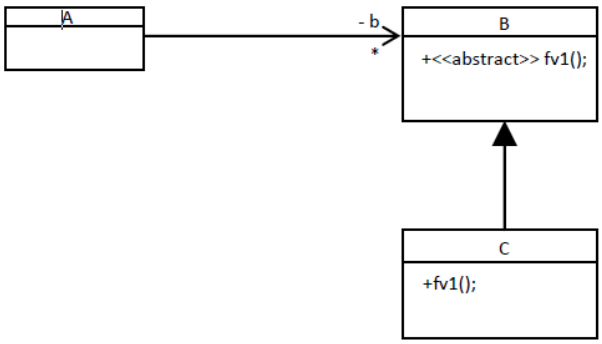
2009. április
Halacs

Szoftvertechnikák PótZH :: 2008-04-30

1. a) Adja meg az alábbi osztálydiagram C# vagy Java kódját (persze ott konkrét nevekkel volt megadva, de a lényeg ez)!




```
class A
{
      private List<B> b;
}

class B
{
      abstract public void fv1();
}

class C : B
{
      public void fv1();
}
```

b) Adja meg, hogy egy objektum elérése interfészen keresztül vagy absztrakt ősosztály használatával milyen előnyökkel illetve hátrányokkal jár!

- **egy osztály több interfészt is implementálhat ⇔ egy osztálynak csak egy őse lehet** (kivéve C/C++-ban ahol több is)

  Ezt olyankor szeretjük, ha van több különböző osztályunk, amelyeknek vannak azonos nevű fv-eik. Ezekkel a fv-ekkel heterogén kollekció alkotható. Abban az esetben, ha több ilyen „közö fv csoport" is van különböző osztályoknál, több interfész használatával, különböző viselkedési módok alapján szervezhetünk kollekciókat anélkül, hogy az interfészben definiált függvényeken túl mást tudnánk az osztályokról.

- interfészesetében ha egy fv-t nem implementáltunk fordítási hibát kapunk ⇔ absztakt ősosztálynál ha egy absztrakt fv-t nem implementálunk az továbbra is absztrakt marad. Csak példányosítási kísérletnél kapunk fordítási hibát. Addig rejtve marad.

2. Ismertesse a C# delegate fogalmát, használatát (nem kell hosszú kód)!

A delegátok típusos metódusreferenciákat jelentenek .NET-ben. Megfelelnek a C nyelv típusos függvény pointereinek. Egy delegát definiálásával egy olyan változót definiálunk, amellyel rámutathatunk egy olyan metódusra, amely típusa (paraméterlistája és visszatérési értéke) megfelel a delegát típusának. A delegát meghívásával az értékül adott (beregisztrált) metódus automatikusan meghívódik. A delegátok használatának egyik előnye az, hogy futási időben dönthetjük el, hogy több metódus közül éppen melyiket szeretnénk meghívni.

Egy delegáttípus létrehozása a delegate kulcsszó használatával történik a következőképpen:

```
public delegate void PrintMessageDelegate(string message);
```

Egy delegát példány definiálása:

```
public PrintMessageDelegate dlg;
```

A delegát példány ráállítása egy metódusra (amely típusa megfelel a delegáttípusnak):

```
...
dlg = new PrintMessageDelegate(PrintMessage);
...

public void PrintMessage(string message)
{
      Console.WriteLine(message);
}
```

A delegát meghívása:

```
dlg("Life is good!");
```

3. a) Mi az az érvénytelen terület? Hogyan kapcsolódik ez a Paint eseményhez?

Korábban takarásban levő, láthatóvá vált ablakrészek (pl. átméretezés, Z-orderben előbb került az ablak, stb.)

Érvénytelen terület keletkezésekor meghívódik a Paint függvény, mely gondoskodik a terület (ablak) újrarajzolásáról.

b) Írjon olyan C# nyelvű, GDI-t használó kódot, ami a (10,10) koordinátában másodpercenként eggyel növelve megjeleníti egy számláló értékét! (GDI-t használó = nem lehet TextBox-ot, Labelt, stb. használni)

```
int counter = 0;
Timer timer = new Timer();
Timer.Interval = 1000;
timer.Tick += new EventHandler(timer_Tick);
timer.Start();

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

      e.Graphics.DrawString(counter.ToString(), this.Font,
                                          Brushes.Black, 10, 10);
}


void timer_Tick(object sender, EventArgs e)
{
      counter++;
      Invalidate();
}
```

4. a) Mutassa be a Dispose mintát (nem kell hosszú kód)!

A dispose tervezési mintának köszönhetően az objektumok által foglalt erőforrások felszabadítása válik determinisztikussá.

Implementáljuk az IDisposable interfészt → Minden osztály tartalmazzon Dispose() függvényt, melynek törzsében felszabadítjuk az erőforrásokat. Ez explicit hívható és hívandó függvény lesz. A tartalmazott, IDisposable-t implementáló felügyelt objektumokra is hívjunk Dispose-t, hogy a tartalmazott objektumok is fel tudják szabadítani a nem felügyelt erőforrásaikat.
A destruktorban is gondoskodjuk a nem felügyelt erőforrások felszabadításáról, hogy akkor se szivárogjanak el, ha a fejlesztő elfelejtett Dispose-t hívni.

```
public class BaseResource : IDisposable
{
    private IntPtr handle;
    private Component Components;
    private bool disposed = false;

    public BaseResource()
    {
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!this.disposed)
        {
            if (disposing)
            {
                Components.Dispose();
            }
            CloseHandle(handle);
            handle = IntPtr.Zero;
        }
        disposed = true;
    }

    ~BaseResource()
    {
        Dispose(false);
    }

    public void DoSomething()
    {
        if (this.disposed)
        {
            throw new ObjectDisposedException();
        }
    }
}
```

A származtatott osztályra vonatkozó részt, lást a külön fájlban!

b) Hasonlítsa össze a C# generikus megoldását az object-et használó Arraylist megoldással!

Objectetként kezelés problémái:

- Castolni kell (plusz kódot kell írni)

```
ArrayList list = new ArrayList();
list.Add( new Person() );
…
int i = (Person)list[0];
```

- Csak futási időben derül ki, ha hiba van

```
ArrayList list = new ArrayList();
list.Add( new Person() );
…
int i = (int)list[0];
```

- Nincs kikényszerítve, hogy ne keveredjenek az objektumok

```
ArrayList list = new ArrayList();
list.Add( new Person() );
list.Add( 12 ); // Ezt bizony elfogadja
```

- Érték típusoknál be és kidobozolás (teljesítmény)

Ezeket kiküszöböli a .NET generikus megoldása. (Generikus típusok.pdf)

c) Hasonlítsa össze a C++ template-et és a C# generikus megoldását!

C++

- A sablonok fordításkor fejtődnek ki!
- Minden sablonparaméter-kombinációra külön kód generálódik!
- Nagyon hatékony
- Kódburjánzás (code bloat) veszély! ☹
- A sablon forráskódja a felhasználás során a fordításkor rendelkezésre kell álljon! A forráskód védelme nem megoldott! ☹

C#

- Futásidőben készülnek el, nem fordításidőben
- Deklaráláskor (fordítás időben) kerülnek ellenőrzésre, nem futásidőben
- Érték- és referencia típusokkal is működnek
- Teljes futásidejű típusinformáció (reflexió) támogatás
- A JIT fordító a sablon kódot specifikus hivatkozásokra cseréli

(Generikus típusok.pdf)

5. a) Adja meg a többszálú alkalmazások három fő előnyét az egyszálúakkal szemben!

1. jobb processzor kihasználtság
2. nem növekvő átlagos válaszidő (interaktivitás – pl. nem akad le a GUI, szerveralkalmazások)
3. időzítés érzékeny feladatok magasabb prioritású szálon futtathatóak

b) Írjon rövid C# kódot egy új szál indítására!

```
        public MainForm()
        {
            // szál létrehozsa + szál fv (delegate) megadása
            Thread t = new Thread(new ThreadStart(ThreadOp));
            // szál indítása
            t.Start();
        }

        // szál fv.
        private void ThreadOp()
        {
           // Itt fut a szálunk. Ha ez a fv véget ér,
           // a szál is befejeződik
        }
}
```

c) Az alábbi kódrészletet alakítsa át úgy, hogy szálbiztos legyen (ezt itt helyben kellett, tehát csak egy kis kiegészítés kell)!

```
public class Stack<T>
{
  readonly int size; int current = 0; T[] items;
  private object syncroot = new object();

  public void Push(T item) {

        lock (syncroot)
        {
          items[current++] = item;
        }

  }

  public T Pop() {

        lock (syncroot)
        {
             return items[current--];
        }
  }
}
```