Programación orientada a objetos. Examen diciembre 2006.

Jueves 21 de diciembre de 2006.

Sean las siguientes clases y programa:

```
interface Persona
{
    void Caminar();
}

interface Empleado
{
    void Cobrar();
}

interface Joven
{
    void Bailar();
}

class Trabajador : Persona, Empleado
{
    public void Caminar()
    {
    }

    public void Cobrar()
    {
    }

    public void Bailar()
    {
    }
}

class Anciano : Persona
{
    public void Caminar()
    {
    }
}

class Program
{
    static void Main(string[] args)
    {
        /*1*/ Trabajador a = new Trabajador();
        /*2*/ Persona b = a;
        /*3*/ Empleado c = a;
        /*4*/ Joven d = a;
        /*5*/ Persona e = new Anciano();
        /*6*/ Empleado f = e;
        /*7*/ Joven g = e;
    }
}
```

1.1 Indiquen las líneas de código incorrectas.

1.2 ¿Cuáles tipos tiene cada uno de los objetos contenidos o referenciados en las variables definidas en las líneas correctas? Indiquen los nombres de los tipos en cada caso. Consideren solo los tipos definidos en esta pregunta.

1.3 ¿Qué mensajes pueden ser enviados a los objetos contenidos o referenciados en las variables `a` y `e`? ¿Cómo lo saben? Escriban los mensajes. Consideren solo los tipos definidos en esta pregunta.

1.4 ¿Los objetos contenidos o referenciados en qué variables tienen el mismo tipo? Para cada tipo en esta situación indiquen las variables. Consideren solo los tipos definidos en esta pregunta.

1.5 ¿Los objetos contenidos o referenciados en qué variables tienen más de un tipo? Para cada variable en esta situación indiquen los tipos. Consideren solo los tipos definidos en esta pregunta.

Sean las siguientes interfaces. Las interfaces están definidas en `System.Collections` y están copiadas aquí con la documentación del .NET Framework para vuestra comodidad. La interfaz `IEnumerable` ya la conocen.

```
/// <summary>
/// Exposes the enumerator, which supports a simple iteration over
/// a non-generic collection.
/// </summary>
interface IEnumerable
{
    /// <summary>
    /// Returns an enumerator that iterates through a collection.
    /// </summary>
    /// <returns>An IEnumerator object that can be used to iterate
    /// through the collection.</returns>
    IEnumerator GetEnumerator();
}

/// <summary>
/// Defines size, enumerators, and synchronization methods for all
/// nongeneric collections.
/// </summary>
interface ICollection : IEnumerable
{
    /// <summary>
    /// Copies the elements of the ICollection to an Array,
    /// starting at a particular Array index.
    /// </summary>
    /// <param name="array">The one-dimensional Array that is the
    /// destination of the elements copied from ICollection. The
    /// Array must have zero-based indexing. </param>
    /// <param name="index">The zero-based index in array at which
    /// copying begins.</param>
    void CopyTo(Array array, int index);

    /// <summary>
    /// Gets the number of elements contained in the ICollection.
    /// </summary>
    Int32 Count { get; }

    /// <summary>
    /// Gets a value indicating whether access to the ICollection
    /// is synchronized (thread safe).
    /// </summary>
    Boolean IsSynchronized { get; }
```

```
        /// <summary>
        /// Gets an object that can be used to synchronize access to
        /// the ICollection.
        /// </summary>
        Object SyncRoot { get; }
    }
```

Un `Bag` es una colección de objetos que no tiene repetidos: si agrego el mismo objeto por segunda vez no ocurre nada. Sean las siguientes clases y programa:

```
using System;
using System.Collections;
using System.Diagnostics;

class Bag
{
    private ArrayList elements = new ArrayList();
    private Int32 limit;
    public Int32 Limit { get { return limit;  } }
    public Bag(Int32 limit)
    {
        this.limit = limit;
    }
    public Boolean Contains(Object element)
    {
        return (elements.IndexOf(element) != -1);
    }
    public void Add(Object element)
    {
        /* Completen el código de este método. */
    }
    public void Remove(Object element)
    {
        /* Completen el código de este método. */
    }
}

class Program
{
    static void Main()
    {
        Bag b = new Bag(2);
        String s = "abc";
        b.Add(s);
        Console.WriteLine(b.Count);
        b.Add(s);
        Console.WriteLine(b.Count);
        b.Add("xyz");
        Console.WriteLine(b.Count);
    }
}
```

2.1 Sabiendo que las colecciones son objetos del tipo `ICollection` hagan las modificaciones necesarias en la clase `Bag` para que las instancias de esta clase tengan el tipo `ICollection`. Tengan en cuenta que las instancias de la clase `ArrayList` también son del tipo `ICollection`.

2.2 Completen los métodos `void Add(Object)` y `void Remove(Object)`.

2.3 Agreguen usando `Debug.Assert(Boolean)` lo que haga falta para especificar las siguientes afirmaciones:

- La cantidad de elementos en el `Bag` es siempre menor que el límite.

- La cantidad de elementos en el `Bag` es siempre mayor o igual que cero.
- Al agregar un elemento, si ese elemento ya fue agregado antes, no cambia la cantidad de elementos en el `Bag`; de lo contrario, la cantidad de elementos aumenta en uno.
- Al agregar un elemento, el `Bag` contiene el elemento.
- Para remover un elemento, el `Bag` debe contener el elemento.
- Al remover un elemento, el `Bag` ya no contiene el elemento.
- Al remover un elemento, la cantidad de elementos en el `Bag` disminuye en uno.

No es necesario que escriban el programa nuevamente, mientras quede bien claro dónde harían los cambios.

2.4 Cambien, agreguen o quiten una línea del código anterior, para que se viole una precondición. Escriban la línea, si corresponde, e indiquen claramente cuál cambia o dónde la agregan o la quitan, en qué método, de qué clase.

2.5 ¿La línea de código de la pregunta anterior podría estar en la clase `Bag`? ¿Y en la clase `Program`? En cada caso indiquen "sí" o "no" y justifiquen la respuesta.

2.6 Cambien, agreguen o quiten una línea del código anterior, para que se viole una poscondición. Escriban la línea, si corresponde, e indiquen claramente cuál cambia o dónde la agregan o la quitan, en qué método, de qué clase.

2.7 ¿La línea de código de la pregunta anterior podría estar en la clase `Bag`? ¿Y en la clase `Program`? En cada caso indiquen solamente "sí" o "no".

Sean las siguientes clases:

```
public class Ancestro
{
    public void Método()
    {
        Console.WriteLine("Yo soy el ancestro");
    }
}

public class Sucesor: Ancestro
{
    public void Método()
    {
        Console.WriteLine("El sucesor soy yo");
    }
}

public class Program
{
    public static void Main()
    {
        Ancestro b = new Sucesor();
        b.Método();
    }
}
```

3.1 ¿Qué aparece impreso en la consola si el lenguaje usa encadenamiento estático?

3.2 ¿Qué aparece impreso en la consola si el lenguaje usa encadenamiento dinámico?

3.3 ¿Qué cambio podrían hacer en el método `Main` de la clase `Program` para que apareciera impreso `"Yo soy el ancestro"` tanto si el lenguaje usara encadenamiento dinámico como estático? Escriban nuevamente la clase `Program`.

3.4 ¿Qué cambio podrían hacer en el método `Main` de la clase `Program` para que apareciera impreso `"El sucesor soy yo"` tanto si el lenguaje usara encadenamiento dinámico como estático? Escriban nuevamente la clase `Program`.

4. Una clase cualquiera puede heredar de una clase abstracta o implementar una interfaz. Desde el punto de vista de los tipos ¿qué similitudes y diferencias existen en ambos casos? Programen un pequeño ejemplo en C#, usando las clases provistas anteriormente y modificándolas según sea necesario, para mostrar al menos una similitud y una diferencia.

Sean las siguientes clases:

```
public class Corredor
{
    public void Correr()
    {
        /*…*/
    }
}

public class Trabajador
{
    public void Trabajar()
    {
        /*…*/
    }
}


public class Programa
{
    public static void Main()
    {
        CorredorProfesional o = new CorredorProfesional();
        o.Correr();
        o.Trabajar();
    }
}
```

5. Escriban la clase `CorredorProfesional` que falta, de forma tal que escriban lo menos posible, pero que el código todavía compile, asumiendo que el lenguaje tiene o no herencia simple o múltiple, según las siguientes opciones:

	Tiene herencia simple	Tiene herencia múltiple
5.1	Sí	Sí
5.2	Sí	No
5.3	No	Sí
5.4	No	No

Si dos opciones tienen la misma respuesta, no escriban nuevamente la clase, sólo indíquenlo.

6. Un objeto se manda un mensaje a sí mismo. ¿El método que se ejecuta como consecuencia tiene que estar implementado en la clase de ese objeto? Justifiquen la respuesta en forma breve y concreta. Muestren la respuesta en un programa de ejemplo si fuera posible; pueden usar las clases que aparecen anteriormente.

7. ¿Puedo prescindir del **encadenamiento estático** en un lenguaje de programación orientada a objetos? ¿Y del **encadenamiento dinámico**? ¿Qué se pierde en cada caso? Respondan en forma breve y concreta.

8. ¿Puede haber **encadenamiento estático** cuando el tipo de una variable está declarado en una interfaz? Justifiquen la respuesta en forma breve y concreta. Muestren la respuesta en un programa de ejemplo si fuera posible; pueden usar las clases que aparecen anteriormente.

Sean las siguientes clases:

```
public class Edit
{
    private string text;
    public string Text
    {
        get { return text; }
        set { SetText(value); }
    }

    protected virtual void SetText(string value)
    {
        text = value;
    }
}

public class Beeper
{
    public void Beep()
    {
        //...
    }
}
```

La clase `Edit` representa un control de edición de un *framework* de interfaz de usuario. La propiedad `Text` representa el texto mostrado en el control. Cuando el usuario escribe en el control el *framework* usa la propiedad `Text` para cambiar el contenido del control. También el programador puede usar la propiedad `Text` para cambiar el contenido del control por código.

La clase `Beeper` tiene la responsabilidad de emitir un sonido "beep".

9. Terminen de programar la siguiente clase de tres formas: la primera usando herencia simple y composición y delegación, la segunda usando herencia múltiple -asuman por un instante que C# soportara herencia múltiple- y la tercera usando solo composición y delegación. La clase `BeeperEdit` debe tener por lo menos los mismos métodos que la clase `Edit`. Los controles `BeeperEdit` deben emitir un sonido "beep" cuando se modifica el texto del control.

```
public class BeeperEdit
{
    // Programen la clase
}
```