Clase 35bis

Martes, 5 Diciembre 2023

Indice

Clase correccion Examen

•Correcciones generales y notas importantes.

•Un método 'Get' no modifica ni el contenido de las listas, ni los valores de un tipo. **Simplemente devuelve un contenido de datos.**

LOS MÉTODOS GET NO MODIFICAN LOS DATOS

•Es posible crear un comportamiento establecido donde una función de objeto llame a una función de clase. Entonces si se puede repetir la función.

```
'
public void CalculateSum()
{
    Function.CalculateSum();
}

public static int CalculateSum(int value1, int value2)
{
    return value1 + value2;
}
'
```

•Una clase llamada 'Prueba' no debe contener una instancia 'Prueba'.

```
'
public class Prueba
{
    private Prueba prueba1;
    private List<Prueba> list;
    Prueba prueba1 = new Prueba();
    // Ninguna de estas líneas debe estar dentro de la clase.
}
'
```

•Prohibido el siguiente código:

```
'
public int GetMethod()
{
      if(funcion)
            return true;
      return false;
}
'
```

Se puede y debe hacer en una línea.

•***ES REALMENTE IMPORTANTE REALIZAR VALIDACIONES DE NUESTROS PARÁMETROS***

•Las colecciones de datos no se devuelven. Si hacemos los metodos **privados**, no es lógico dar el control de la lista al usuario.

Lo ideal es **crear las funciones que necesite el usuario para utilizar nuestra clase**. Somos nosotros los que decidimos como se utiliza nuestra clase.

Clase 36

Lunes, 11 Diciembre 2023

Indice

Primera clase de Herencia

- Si tengo las siguientes clases:

```
'
public class Animal
{
    public string color;
}

public class Vegetal
{
    public string color;
}
'
```

El atributo 'color' es código repetido. Esto nos indica que podemos crear una clase **'Padre'** que contenga el atributo común 'color'. Es de esta clase de la que **'Heredan'** las clases hijas.

```
'
public class SerVivo
{
    public string color;
}

public class Animal : SerVivo
{
    // contiene 'color' por herencia.
}

public class Vegetal : SerVivo
{
    // contiene 'color' por herencia.
}
'
```

Ser Vivo es una clase que contiene los elementos comunes a las clases 'hijas'.

Ventajas de la Herencia

•El código en común se escribe en la clase Padre. No tenemos código repetido.
•El siguiente código es correcto:

```
'
SerVivo a1 = new SerVivo();
Animal a2 = new Animal();
Vegetal a3 = new Vegetal();
'
```

El siguiente código **NO es correcto.**

```
'
Animal a4 = new SerVivo();
Vegetal a5 = new SerVivo();
'
```

a4 y a5 son de tipo 'Hija' y la clase 'Padre' no contiene todos los elementos implementados en sus clases hijas.
Es decir SerVivo no contiene los métodos de las clases hijos.
•El código es incorrecto porque la clase **Padre: SerVivo** no contiene los métodos de la clase **Hijo: Vegetal / Animal**
•El siguiente código es correcto:

```
'
SerVivo a6 = new Animal();
SerVivo a7 = new Vegetal();
List<SeresVivos> list; // Puede contener SeresVivos, Animales o Vegetales.
'
```

Interfaces

•Las clases, **que no contienen atributos** y todos sus métodos están *declarados*, pero no *definidos,* reciben el nombre de ***Interfaces***.

```
'
public interface Colegio
{
    void CreateBuilding();
    void GetStudentsCount();
    void GetStudentAtClassroom(int class);
}
'
```

Clase 37

Martes, 12 Diciembre 2023

Indice

Segunda clase de Herencia

•Existen distintas maneras de inicializar objetos. Los constructores es una de esas maneras

```
'
public class Person
{
	public string _name;
	public Gender gender;

	public Person()
	{

	}

	public Person()
	{
		_name = ""
		_gender = Gender.Male
	}

	public Person(string name, Gender gender)
	{
		_name = name;
		_gender = Gender;
	}

	public Person() : this.(string name, Gender gender)
	{

	}
}
'
```

•Estos son distintos constructores. Podemos inicializar los valores en linea o a través de los distintos constructores.
•Podemos crear una subclase que contenga propiedades de la clase superior y que además contenga sus métodos. Para ello empleamos la Herencia.
•

```
'
```

```
public class Student : Person
{

}
public class Teacher : Person
{

}
'
```

•Un constructor no debe realizar ni cálculos ni calcular Randoms. Aquí unicamente se inicializan objetos.
•La clase **Base** es la clase de la que heredo
•Puedo llamar a otro constructor y pasarle por parametros los valores que requiere.

```
public Teacher (string name, Gender gender, int Bloodlust) :
base(name, gender)
{
    _bloodlust = Bloodlust;
}
```

•La superclase de la cual heredan todas las clases es **'Object'**. Aunque no aparece explicitamente en el código, el compilador lo incluye automáticamente.
•El código 'base' siempre llama a la clase superior.
•Es sensible al orden y nombre de los parámetros.
•

```
'
public Person (string name, Gender = Gender.MALE)
{

}
'
```

Puedo establecer un valor por defecto. En ese caso no es necesario indicarlo en los parámetros.
•Existe una manera de realizar el inicializador de valores desde la propia creacion 'new' del objeto.

```
'
public static string[] Main(string[] args)
```

```
{
    Teacher teacher1 = new Teacher();

    Teacher teacher2 = new Teacher()
    {
        _name = "Juan";
        _gender = Gender.MALE;
    };

    Teacher teacher3 = new Teacher(){"Juan",
Gender.MALE};
}
'
```

•La herencia **no se aplica a funciones de clase.**La herencia solo funciona con clases de objeto, es decir, con aquellos elementos que realizan un 'new' para ser creados

Properties

•Funciones especialmente diseñadas para C# que nos permite simplificar los métodos para coger una función y establecer un valor.

```
'
public string Name
{
    get
    {
        return _name;
    }
    set
    {
        _name = value;
    }
}
'
```

•Podemos simplificar de varias maneras este codigo fuente.

```
'
public int PersonCount
{
    get
    {
```

```
            return list.Count;
        }
    }

    public int GetPersonCount()
    {
        return list.Count
    }

    public int PersonCount
    {
        get
        {
            GetPersonCount();
        }
    }
    '
```

•No se deben realizar validaciones dentro de las properties. A excepción de código sencillo. También es bueno crear funciones auxiliares que hagan mas sencilla la tarea de nuestros compañeros.

Notas

•Debemos darnos cuenta que nuestro trabajo consiste en crear funciones que utilizarán el resto de compañeros. Somos nosotros los que indicamos 'cómo' deben utilizar nuestro trabajo, además de añadir las funciones que sean necesarias para trabajar.
•Las funciones básicas para trabajar con una lista son:

GetListCount()
GetElementAt(int index)
AddElement(Element element)
RemoveAt(int index)

Clase 38

Jueves, 14 Diciembre 2023

Indice

Tercera clase de Herencia

•La herencia sirve para realizar una jerarquía de clases. Es un método rápido si tenemos que extraer datos de una serie de clases.
•Su uso se optimiza si la petición de datos se hace de forma repetitiva.
•No es útil si los datos que vamos a consultar tienen una duración corta en el tiempo. Por ejemplo, con aplicaciones tipo Whatsapp no es eficiente.

Nota sobre el funcionamiento de la herencia

•La instrucción:

```
Person person1 = new Teacher();
```

Crea un **Objeto** de tipo **Teacher** dentro de un puntero de tipo **Person** que apunta a la direccion de la memoria donde se almacena dicha instancia.
•Cuando trabajamos con objetos estamos constantemente almacenando en memoria RAM las cualidades de un objeto, hacia donde apunta el puntero de la variable.
•**El tipo no contiene NINGÚN DATO. Solo contiene un*número* que apunta a la dirección de memoria donde se encuentran los atributos y cualidades del objeto**
•La variable 'person1' solo contiene un numero.

Nuevo funcionalidad de las clases

•Si tenemos el siguiente metodo dentro de una clase, por ejemplo Person:

```
'
public class Person
{
    public string GetFullName()
    {
        return _name;
    }
}
'
```

•Este método nos devuelve el nombre del objeto.
•Ahora voy a permitir que cualquier hija suya pueda usar esta función

```
'
public class Person
{
    public virtual string GetFullName()
    {
        return _name;
    }
}

// la instrucción virtual indica que la función puede ser definida en las clases hijas

public class Teacher
{
    public override string GetFullName()
    {
        return ">PROFESOR< + Name + >/PROFESOR<"
    }
}

// La instrucción override indica que el método va a ser sobrescrito por la clase hija.
'
```

Notas prácticas

•A la hora de saber cual es la función que tiene que utilizar, el compilador se dirige **a la última función con instruccion 'override', dentro de la jerarquía de clases**
•No es obligatorio que todas las clases hijas descendientes sobreescriban la funcion virtual.
•Todos los objetos contienen en su interior un 'vtable' que indica al compilador cual es la última de las funciones 'verride'
•Si el objeto creado es ' new Teacher()' busca dentro de teacher el último metodo override hasta su la superclase padre
•Si fuera de tipo 'new Director()' buscaría desde Director hasta Person.
•Existe la posibilidad de realizar un 'casting' sobre 'Person' para transformalo en 'Teacher'.
•No se puede acceder a los atributos de clases hijas, pero Sí se puede acceder a sus métodos desde la notación por punto

•**VIRTUAL:** Invoca al último método definido con **'OVERRIDE'**
•**Base.Metodo():** Invoca a la última clase superior donde se define el método.
•'Protected': Permite el acceso a las clases hijas. Es una función pública dentro de la jerarquía de clases. Es una función privada fuera de la jerarquía de clases. Los Métodos también pueden ser 'protected'

Herencia por obligación

•Si quiero **Obligar** a que los hijos implementen alguna función, tengo que utilizar la instrucción **Abstract.**
•En el mismo momento que se indica una función 'abstract' es obligatorio que las hijas implementen la función. Además la clase se convierte en una clase 'abstract' de la cual está prohibido realizar instancias.

Funciones puramente abstractas

•La siguiente clase es puramente abstracta:

```
'
public class abstract Perro
{
    public abstract void Metodo1();
    public abstract void Metodo2();
    public abstract void Metodo3();
    public abstract void Metodo4()
}
'
```

•No contiene atributos
•Todos sus métodos son abstractos
•A este tipo de clases s les denomina **'Interfaces':**
•Una interfaz define las directivas que debe cumplir una función para poder heredar de esta interfaz

```
'
public interface IPerro
{
    void Metodo1();
    void Metodo2();
    void Metodo3();
    void Metodo4();
}
'
```

•Por reglas de estilo se añade la letra mayúscula 'I' Justo delante de su nombre.
•Si la clase hija implementa todos sus métodos, entonces puede heredar de la interfaz.
•Se utiliza para definir la estructura de un programa.

Clase 39

Viernes, 15 Diciembre 2023

Indice

Ejercicio Propuesto

•Crear la jerarquía de clases para un programa de diseño automático de dibujo vectorial.

Clase: Crea un punto 2D.

POINT2D

+x : double
+y : double

Interfaz: Directiva de estructura Blueprint.

<<IBlueprint>>

+AddShape(Shape: IShape)
+GetShapeCont(): int
+GetShapeAt(int index)
+RemoveAt(int index)
+GetArea(): double

Clase: Coleccion de Shapes.

Blueprint

-List<IShape>

Interfaz: Directiva de estructura Shape.

<<IShape>>

+GetPosition()
+SetPosition()
+GetArea(): double
+HasArea(): bool
+GetShapeType(): shapeType
+GetName(): string
+SetName(string: name)

Clase: Forma o Shape definida.

Shape

#position: Point2D
_name: string

Clase: Shape *sin* Area.

ShapeWithoutArea

Clase: Shape *con* Area.

ShapeWithArea

Clase: Segment.

Segment2D

_from
_to

Clase: Rectangle

Rectangle

_min: Point2D
_max: Point2D

+SetRectangle(Point2D point2D)
+GetRectangle: Point2D
+GetMin: Point2D
+GetMax: Point2D
+SetMax(Point2D)
+SetMin(Point2D)
+GetWidth(): double
+GetHeight(): double

Clase: Circle

Circle

_radius: double
_center: Point2D

+SetCircle(Point2D point2D)
+GetCircle(): Point2D

Clase: Polyline.

Polyline2D

+AddPoint(Point2D point2D)
+GetPointCount(): int
+GetPointAt(int index): Point2D
+SetPointAt(P)

Clase 40

Lunes, 18 Diciembre 2023

Indice

Correccion de la clase 'Planos'

•Utilizaremos un 'enum' si los valores están muy definidos, y la lista de tipos no parece que vaya a cambiar en el tiempo.
•Si los valores del 'enum' cambian, es mejor utilizar otro recurso.
•**JAMÁS devolveremos una lista o colección de datos(arrays, arboles, diccionarios).** Esto se debe a que si devolvemos una colección, pueden modificar de manera negativa nuestros valores.
•**PROHIBIDO PUBLIC LIST<type> GETLIST**
•La función **ToString** es **MUY** aconsejable. Ya existe una definición de dicha función, es por ello que hace falta un 'override'. Devuelve un string de datos.
•La función **Equals** ya está definida en el lenguaje, pero tiene un comportamiento no del todo acertado. Puede ser de tipo *superficial*, si los dos objetos apuntan al mismo puntero, o de tipo *profundidad* si tiene en cuenta **todos** los parámetros.

Clase 41

Martes, 19 Diciembre 2023

Indice

Funciones Delegados y Funciones Lambda

•Estas funciones nos ayudan a simplificar nuestro código. Son expresiones más sencillas que normalmente se resuelven en una línea. Su misión principal es mejorar la legibilidad del código y ahorrarnos trabajo en escritura.
•Las funciones Lambda son **simplificaciones** de las llamadas que hacemos a las funciones. De tal manera que las funciones Lmabda se suelen resolver en una única línea de código, evitando la escritura de la función que simplifica.
•Por ejemplo:
'Imprimir (() => { })'
Este código imprime, la función que recibe 0 parametros'()' y cuyo cuerpo de función se encuentra entre '{}'
•La vida útil de esta función ocurre simplemente en el momento de compilar 'Imprime', luego desaparece del programa.
•Habitualmente estas funciones contienen un **único atributo, un único método y un único constructor**
•Los {} pueden omitirse si el cuerpo de la función ocupa una línea.

Clase 42

Jueves, 21 Diciembre 2023

Indice

Correcciones Herencia

•Es posible utilizar la propiedad 'read-only' para evitar la escritura de un atributo. Lo establece como si fuera una constante, un 'prefetch' o ajuste predefinido.
•Respetar el orden de las comprobaciones de rango. Si queremos escribir, '0 < a < 10', respetamos la posición de la variable entre los extremos, '0 < a && a < 10'
•No olvidarse de validar los parámetros de entrada.
•Repaso de punteros.
•Clase repaso de números mágicos. Son valores que aparecen en el código de improviso, sin ningún tipo de comentario. Esto puede producir errores y bugs sin resolver, lo conveniente es **SIEMPRE** declarar y definir el número dentro de una variable.

Clase 43·A

Domingo, 24 Diciembre 2023

Indice

Delegados

•Ejemplos practicos numero 1

```
'
public class Bienvenida
{
    public static void MensajeBienvenida()
    {
        Console.WriteLine("Este es un mensaje de bienvenida");
    }
}

public class Despedida
{
    public static void MensajeDespedida()
    {
        Console.WriteLine("Este es un mensaje de despedida");
    }
}

public delegate void MetodoDelegado()

public class program
{
    public static Main()
    {
        MetodoDelegado delegado1 = new
MetodoDelegado(Bienvenida.MensajeBienvenida)
        delegado1();

        MetodoDelegado delegado2 = new
MetodoDelegado(Despedida.MensajeDespedida)
        delegado2();
    }
}
'
```

•Ejemplos practicos numero 2

```
'
public class Bienvenida
{
    public static string MensajeBienvenida(string text)
    {
        return string.Format($"Este es un mensaje de
bienvenida: {0}",text);
    }
}

public class Despedida
{
    public static string MensajeDespedida(string text)
    {
        return string.Format($"Este es un mensaje de despedida:
{0}",text);
    }
}

public delegate string MetodoDelegado(string message)

public class program
{
    public static Main()
    {
        MetodoDelegado delegado1 = new
MetodoDelegado(Bienvenida.MensajeBienvenida)
        string result1 = delegado1("Bienvenido de nuevo");
        Console.WriteLine(result1);

        MetodoDelegado delegado2 = new
MetodoDelegado(Despedida.MensajeDespedida)
        string result2 = delegado2("Hasta la proxima, Ecos");
        Console.WriteLine(result2);
    }
}
'
```

•Ejemplos practicos numero 3

```
'
public delegate void DelegadoImprimir(string message)

public class Imprimir
{
```

```
                    public void MensajeBienvenida(string text)
                    {
                        Console.WriteLine(text);
                    }

                    public void MensajeDelegado(string text)
                    {
                        DelegadoImprimir delegado = new
DelegadoImprimir(MensajeBienvenida);
                        delegado(text);
                    }
                }

                public class program
                {
                    public static Main()
                    {
                        Imprimir test = new Imprimir();
                        test.MensajeDelegado("Este es el texto que se imprime
por consola");
                    }
                }
                '
```

Clase 43·A

Domingo, 24 Diciembre 2023

Indice

Delegados

•Ejemplos practicos numero 1

```
'
public class Bienvenida
{
    public static void MensajeBienvenida()
    {
        Console.WriteLine("Este es un mensaje de bienvenida");
    }
}

public class Despedida
{
    public static void MensajeDespedida()
    {
        Console.WriteLine("Este es un mensaje de despedida");
    }
}

public delegate void MetodoDelegado()

public class program
{
    public static Main()
    {
        MetodoDelegado delegado1 = new
MetodoDelegado(Bienvenida.MensajeBienvenida)
        delegado1();

        MetodoDelegado delegado2 = new
MetodoDelegado(Despedida.MensajeDespedida)
        delegado2();
    }
}
'
```

•Ejemplos practicos numero 2

```
'
public class Bienvenida
{
    public static string MensajeBienvenida(string text)
    {
        return string.Format($"Este es un mensaje de
bienvenida: {0}",text);
    }
}

public class Despedida
{
    public static string MensajeDespedida(string text)
    {
        return string.Format($"Este es un mensaje de despedida:
{0}",text);
    }
}

public delegate string MetodoDelegado(string message)

public class program
{
    public static Main()
    {
        MetodoDelegado delegado1 = new
MetodoDelegado(Bienvenida.MensajeBienvenida)
        string result1 = delegado1("Bienvenido de nuevo");
        Console.WriteLine(result1);

        MetodoDelegado delegado2 = new
MetodoDelegado(Despedida.MensajeDespedida)
        string result2 = delegado2("Hasta la proxima, Ecos");
        Console.WriteLine(result2);
    }
}
'
```

•Ejemplos practicos numero 3

```
'
public delegate void DelegadoImprimir(string message)

public class Imprimir
{
```

```
                    public void MensajeBienvenida(string text)
                    {
                        Console.WriteLine(text);
                    }

                    public void MensajeDelegado(string text)
                    {
                        DelegadoImprimir delegado = new
DelegadoImprimir(MensajeBienvenida);
                        delegado(text);
                    }
                }

                public class program
                {
                    public static Main()
                    {
                        Imprimir test = new Imprimir();
                        test.MensajeDelegado("Este es el texto que se imprime
por consola");
                    }
                }
                '
```

Clase 44

Lunes, 8 Enero 2024

Indice

Genéricos

•Podemos establecer que una clase trabaja con **valores genéricos** desde su propia definición en el programa.
•Por convencion se utiliza la letra mayúscula **T** para indicar que son valores genéricos.

```
'
public class Ejemplo<T>
{
    // Esta clase funciona con genéricos
}
'
```

•Normalmente utilizamos genéricos si vamos a trabajar con **colecciones**.
•Las colecciones en C# son las siguientes:
Listas, Arrays, Diccionarios, Stack, Queue, Tree, Set.
•Si la clase que usamos es de tipo Genérico, debemos indicar en la instanciación del objeto, cual es el tipo que reemplaza el valor genérico

```
'
Ejemplo<double> ejemplo1 = new Ejemplo<double>();
// El generico T se reemplaza por un tipo double.
'
```

Colecciones I: Stack

•Es una colección de datos donde se apilan los elementos que vamos añadiendo y se extrae **SIEMPRE** el último dato que hemos introducido.
•Es de tipo LIFO: Last In First Out
•Nunca se eliminan elementos.
•Nunca se ordena la lista.

STACK
****
_stack
****
+ Push(element): void
+ Pop(): element
+ Top(): element
+ IsEmpty: bool
+ GetCount(): int

•Existe una instruccion que establece un valor por defecto para los valores de Genéricos, 'default(T)'.

Clase 45

Martes, 9 Enero 2024

Indice

Properties

•Hasta ahora las properties devolvían el valor de un atributo (getter) o establecían el valor de un atributo (setter)
•Se trataba de funciones creadas por el propio programador

```
'
public class Student
{
    private string _name;

    public string GetName()
    {
        return _name;
    }

    public void SetName(string value)
    {
        _name = value;
    }
}
'
```

•A partir de ahora vamos a utilizar el método convencional del lenguaje C#.
•Existen varios modos de crear las properties.
•El más completo es el siguiente:

```
'
    public string Name
    {
        get
        {
            return _name;
        }

        set
        {
            _name = value;
        }
    }
'
```

•Las properties se definen con PascalCase
•La función no recibe parámetros, pero si dispone de cuerpo.
•Si el cuerpo es demasiado grande, es aconsejable sacar el código en una función aparte.
•Otra forma más sintética es la siguiente:

```
'
public string Name
{
	get => _name;
	set => _name = value;
}
'
```

•Por último, si el atributo solo tiene un getter:

```
'
public string Name => _name;
'
```

•También existen los siguientes casos, cuyo uso depende del programador.

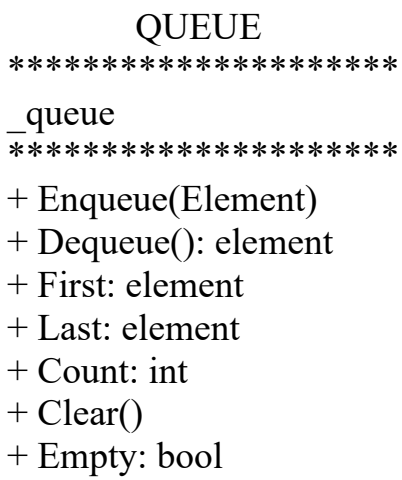
```
'
public string Name
{
	get { return _name;}
	set { _name = value;}
}

public string Name
{
	get;
	set;
}
'
```

•En este último ejemplo, de forma invisible el Lenguaje está creando un atributo relacionado con este getter, pero no lo estamos controlando personalmente-

Colecciones II: Queue

•Es una colección de datos donde se encolan los elementos que vamos añadiendo, de tal forma que siempre sale el primero que ha entrado.
•Es de tipo FIFO: First In First Out
•Nunca se eliminan elementos.
•Nunca se ordena la colección.
•Diagrama de caso de uso



•Normalmente, una función recibe el nombre siguiendo el patrón *Verbo+sustantivo* o simplemente *Verbo*
•Una property sigue el patrón de *Sustantivo* y se escribe sin parámetros.
•Existe una instruccion que establece un valor por defecto para los valores de Genéricos, 'default(T)'.

Clase 46

Jueves, 11 Enero 2024

Indice

Función **EQUALS**

•La función por defecto de C# realiza una comparación superficial de dos elementos, comprobando que los dos objetos apuntan a la misma referencia.
•Es posible que dos objetos tengan los mismos parámetros pero apunten a dos referencias distintas (distintos 'new') y por tanto la función devuelve que los objetos **no son los mismos**, cuando sus atributos son iguales.
•Esto se arregla haciendo nuestra propia función **Equals**:

```
'
public override bool Equals(Object obj)
{
    if(this == obj)
        return true;

    if(obj is not Student)
        return false;

    Student s1 = (Student)obj;
    if(this.Name == s1.Name)
        return true;
}
'
```

•Se comprueba que apuntan a la misma referencia.
•Se comprueba si coinciden los tipos de datos.
•Se comprueba que todos los atributos son iguales. Esto se llama copia en profundidad.
•Es buena práctica que el programador implemente una funcion 'SetHashCode' siempre que se cambia el 'Equals'.

Colecciones III: Set

•Es un tipo de colección que **no admite duplicados**. Si se le pide un dato que no contiene, **no devuelve nada.**
•Diagrama de caso de uso

SET
****

_set
****
+ Add(Element)
+ Remove(Element)
+ RemoveAt(index)
+ IndexOf(Element)
+ Contains(Element)
+ Empty: bool
+ Count: int

•Siempre que nos pidan una función **Contains**, es buena idea hacer con una función **IndexOf** que nos devuelva el índice del elemento. De esta forma tenemos una función útil, que ademas facilita el **RemoveAt(index)**.
•Un 'operator' nos permite modificar el comportamiento de una operación de cálculo.
' public static bool operator ==(Student? s1, Student? s2) { return true; } '
•No es habitual que una colección 'Set' contenga 'nulls'.

Clase 47

Viernes, 12 Enero 2024

Indice

Significado de una Hash

•Un 'Hash' nos permite simplificar la definicion de un objeto. Por ejemplo, si un objeto es un número, el resto de una division seria el hash.
•Este numero Hash no debe repetirse para elementos diferentes. Cada objeto debe tener un hash propio, si no, no tendria sentido.
•Uno de sus usos mas empleados es para guardar la contraseña de los usuarios de un servidor.
•Si queremos comprobar si un objeto es igual a otro, en lugar de comprobar en profundidad sus atributos, podemos primero comprobar que tengan el mismo Hash. Solo en caso que tengan el mismo numero de referencia, comprobamos su semejanza en profundidad.
•C# contiene por defecto una clase llamada 'HashSet'. Su funcionamiento es el siguiente: comprueba el numero Hash y si coincide ejecuta el metodo 'Equals'.
•Existe una norma que indica que si sobreescribes el Equals por defecto en una clase, es obligatorio sobreescribir el 'GetHashCode'.

Clase 'SetWithHash'

•Crear una clase 'Set' que funcione con un 'Hash'.

Clase 47

Viernes, 12 Enero 2024

Indice

Significado de una Hash

•Un 'Hash' nos permite simplificar la definicion de un objeto. Por ejemplo, si un objeto es un número, el resto de una division seria el hash.
•Este numero Hash no debe repetirse para elementos diferentes. Cada objeto debe tener un hash propio, si no, no tendria sentido.
•Uno de sus usos mas empleados es para guardar la contraseña de los usuarios de un servidor.
•Si queremos comprobar si un objeto es igual a otro, en lugar de comprobar en profundidad sus atributos, podemos primero comprobar que tengan el mismo Hash. Solo en caso que tengan el mismo numero de referencia, comprobamos su semejanza en profundidad.
•C# contiene por defecto una clase llamada 'HashSet'. Su funcionamiento es el siguiente: comprueba el numero Hash y si coincide ejecuta el metodo 'Equals'
•Existe una norma que indica que si sobreescribes el Equals por defecto en una clase, es obligatorio sobreescribir el 'GetHashCode'

Clase 'SetWithHash'

•Crear una clase 'Set' que funcione con un 'Hash'

Clase 47

Viernes, 12 Enero 2024

Indice

Significado de una Hash

•Un 'Hash' nos permite simplificar la definicion de un objeto. Por ejemplo, si un objeto es un número, el resto de una division seria el hash.
•Este numero Hash no debe repetirse para elementos diferentes. Cada objeto debe tener un hash propio, si no, no tendria sentido.
•Uno de sus usos mas empleados es para guardar la contraseña de los usuarios de un servidor.
•Si queremos comprobar si un objeto es igual a otro, en lugar de comprobar en profundidad sus atributos, podemos primero comprobar que tengan el mismo Hash. Solo en caso que tengan el mismo numero de referencia, comprobamos su semejanza en profundidad.
•C# contiene por defecto una clase llamada 'HashSet'. Su funcionamiento es el siguiente: comprueba el numero Hash y si coincide ejecuta el metodo 'Equals'
•Existe una norma que indica que si sobreescribes el Equals por defecto en una clase, es obligatorio sobreescribir el 'GetHashCode'

Clase 'SetWithHash'

•Crear una clase 'Set' que funcione con un 'Hash'

Clase 50

Jueves, 18 Enero 2024

Indice

Significado de una Tupla

•Una tupla es el resultado de devolver varios elementos desde una misma función. Es un retorno de multiples valores.
•Existen distintos métodos, cada uno tiene su ámbito de aplicación.

Tupla: Método 1

•La función especifica cuantos y el tipo de valores que queremos retornar.

```
'
public static (int, string) TuplaFunction()
{
    int Item1 = 5;
    string Item2 = "prueba";

    return (Item1, Item2);
}
'
```

Esta función devuelve un valor tipo int y un string.
•Si no se nombran las distintas variables, por defecto los objetos se nombre con 'Item' y un numero de serie.
•También podemos darle nombre a la tupla:

```
'
public static (int numeroTest, string palabraTest)
TuplaFunction()
{
    numeroTest = 5;
    palabraTest = "prueba";

    return (numeroTest, palabraTest);
}
'
```

Tupla: Método 2

•Creamos una clase totalmente nueva, que contenga tantos atributos como retornos tenga la tupla. Simplemente devolvemos todos sus atributos como si fueran un unico resultado.

```
'
public class Tupla
{
    private int Item1;
    private string Item2;
}
'
```

•Este método es 100% compatible con Java.

Tupla: Método 3

•'Out', 'In' y 'Ref'.
•Utilizamos estos comandos si queremos que un retorno se almacene en la variable que nosotros elegimos especificamente.
•En realidad le estamos pasando la dirección de la memoria donde queremos que se almacene la información.

```
'
public static void Test(out int resultInteger)
{
    resultInteger = 5;
}
'
```

Estamos almacenando en la variable 'resultInteger' el valor deseado.
•Cada comando se utiliza en un momento determinado:
'out': El valor se **escribe** en la variable. 'in': El valor sólo se puede leer desde esa variable. Se debe declarar la variable antes de utilizarla. 'ref': El valor se puede escribir y leer desde la variable.
•Aviso, la tupla consume *Mas* recursos que un 'in' y 'out'.

Clase 51

Viernes, 19 Enero 2024

Indice

Test de unidad

•Con los test de caja blanca tienes acceso al código fuente. En los test de caja negra, solo puedes hacer pruebas y comprobar el resultado de los test.
•Existen diferentes modos de hacer test de caja blanca. Uno de ellos es mediante herencias y clases especificamente dedicadas a comprobar que el resultado es el esperado.

```
'
public class TestResults
{
    public bool Empty {get; set;}
    public int Count {get; set;}
}

public static TetsResults Test1(ISet)
{
    // Aqui se ejecutan las pruebas
}
'
```

•Podemos dirigir los resultados hacia un archivo JSON, de tal manera que podemos comprobar si las salidas de cada test corresponden con lo esperado.
•Para ello se utiliza las funciones dedicadas de C#
'JsonSerializer.Serialize(result);'

Clase 52

Lunes, 22 Enero 2024

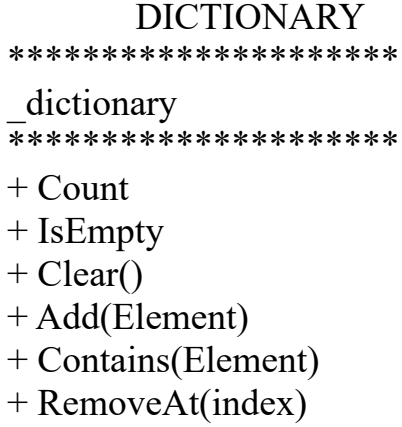
Indice

Diccionarios

•Un Diccionario funciona con pares 'clave / valor' ('<key / value>')
•Solo existe una única clave, no se pueden repetir al igual que los item de un 'Set'.
•Existen distintas opciones para usar keys repetidas:
Se puede machacar el valor repetido.
Se lanza un error, esta es la mejor opcion.
No permito que se sobrescriba el valor.

```
'
public class Diccionario
{
    private class Item
    {
        public K key;
        public V value;
    }
}
'
```

•Diagrama de caso de uso



La función 'Contains' solo busca los keys. La función 'Remove' funciona para eliminar el par key, value.

•Podemos introducir una nueva funcion, la función 'Filter'. La funcion devuelve un diccionario del mismo tipo, cuyo contenido son pares 'clave - valor' que hemos filtrado mediante una condicion de busqueda
•Para ello utilizamos los delegados.
•Un delegado es el **Prototipo** de una función. Un delegado contiene en su interior una función, por lo tanto utilziar un delegado es lo mismo que utilizar una función. La diferencia se encuentra en que es el usuario de nuesta función Filter quien decide las condiciones de la función delegado.
•Un delegado consta de 3 partes diferenciadas:
La definición del delegado
La descripción de la función que utiliza un delegado
La llamada a la función mediante una función lambda.
•El siguiente es un ejemplo de funcion 'Filter' usando un delegado

```
'
// La definición del delegado
public delegate bool DelegateFilter(Item / Content);

// La descripción de la funcion Filter
public List<Tipo> Filter(DelegateFilter del)
{
    List result = new List();
    for(int i = 0; i < list.Count; i++)
    {
        if(del(list[i]))
        {
            result.Add(list[i]);
        }
    }
    return result;
}

// La llamada a la función
Dictionary dictionary = new Dictionary();
dictionary.Filter(item =>
{
    return item == test;
});
'
```

Clase 53

Martes, 23 Enero 2024

Indice

Continuacion de proyecto

•Es importante a la hora de realizar un programa, distinguir entre las funciones que son **públicas** y cualquiera tiene acceso, y las funciones que son **privadas**, y tienen un acceso restringido a nuestro criterio.
•Es preferible mantener privadas las funciones que no va a utilizar nuestro usuario, para evitar malentendidos y peores funcionamientos de la experiencia del programa.
•Una función 'Sort' no crea una colección nueva, no se hace ningún 'new', solo se ordena la colección preexistente.
•Los delegados que utilizan funciones 'Sort' reciben el nombre de 'comparator'.
•Las colecciones se ordenan siguiendo los criterios de la funcion lambda que le pasa el usuario.
•El delegado 'comparator' habitualmente devuelve un int, -1, 0, 1 según el elemento se coloque en la posición anterior, en la misma posición o en la posición siguiente.

Servidores Web

•Existen distintas arquitecturas para el diseño de un servidor:
Java: springboat. *Javascript*: NodeJs. *Go*: Go. *Phyton C#*: ASP.net

Clase 54

Jueves, 25 Enero 2024

Indice

Los 'struct' en C#

•Habitualmente una clase funciona con direcciones de memoria. Las variables de objeto apuntan a una dirección que está en la memoria RAM, que es realmente donde se almacena la información.
•En un 'struct' la información se guarda en la misma 'stack' o memoria de secuencial del programa, que está ordenada según una colección LIFO, se libera de manera secuencial y por lo general el acceso es mucho más rápido.
•El 'stack' no puede crecer, es de un tamaño establecido. Si se completa toda su memoria, se produce un *StackOverflow*. Los datos se almacenan de forma desordenada, según la memoria se ha ido liberando.
•Los archivos que se crean en el 'stack' son de tipo *value types.*
•El 'heap' no tiene un tamaño establecido y los datos se colocan de manera desordenada. Cada objeto apunta a una referencia, que es la dirección donde esta almacenada su información.
•Los elementos que se crean en el 'heap' son de tipo *reference type.*
•Un 'struct' es una clase con atributos y funciones que se guardan directament en el stack del programa, son de tipo value type y no guardan dirección de memoria.
•Sus ventajas son que su acceso es más rápido y directo, además es más económico en materia de rendimiento. Entre sus desventajas está el hecho que no soportan herencia ni interfaces.

Clase tipo 'record'

- Un 'record' es una clase cuyos valores son de 'read-only'. Es inmutable y no permite la herencia.

```
'
public record Student(string Name, int Age)
{

}
'
```

- Los parámetros Name y Age no cambian durante el funcionamiento del programa.

Clase 55

Viernes, 26 Enero 2024

Indice

Repaso 'Heap' vs ' Stack

•Un 'struct' permite mucha mayor cantidad de datos de fácil acceso. Además estos datos tienen una forma muy estructurada.
•Un 'struct' no tiene herencia ni interfaces

Garbage Collector

•Cuando el elemento 'A' no tiene ningún puntero o referencia, el obejto se destruye.
•El runtime de C# no borra los elementos que se guardan en el 'Heap', sino que los mete en una lista de objetos 'muertos'
•Se trata de un comportamiento de 'Pull', donde los objetos desechados se guardan en una lista aparte, una minicaché de elementos muertos.
•Esta caché recibe el nombre de 'Garbage Collector'
•Aun asi, con una funcion que limpie el garbage collector, tampoco te aseguras que se destruya por completo el objeto.
•No puedes programar el destructor porque no sabes exactamente cuando se destruye el objeto.
•Garbage Collector es la lista de objetos muertos.
•El algoritmo 'LRU' o 'less recently used' elimina los elementos que mas tiempo llevo sin utilizar.

TREE

•Contenedor de manera que un nodo es el principal o 'root', del cual crecen el resto de nodos de forma jerarquica descendiente.
•El primero es el nodo 'root'.
•Cada nodo tiene una lista de nodos, que son sus hijos o 'children'.
•Todos los children tienen el mismo 'root'
•Los atributos parent y children son privados. El padre del root es NULL.
•Los elementos nodos que no tienen hijos se llaman hojas o 'leaf'
•Las listas son privadas, nunca se devuelven.
•Level es la posición en la estructura del árbol.
•La función Remove es privada.

Clase 56

Lunes, 29 Enero 2024

Indice

Continuación de la clase 'Tree'

- Clase continuación de 'Tree'.

Clase 57

Martes, 30 Enero 2024

Indice

Continuación de la clase 'Tree'

•Creamos una función que encuentra un elemento entre los nodos hijos. Para ello utilizo un delegado, con los parámetros de búsqueda.

```
'
Node<T> findNode (DelegateChecker checker)
'
```

•Definición completa de la función de búsqueda de un nodo
•Función 'Filter'
•Definición del delegado.

```
'
public delegate bool DelegateFilter(Node<T>);
'
```

•Definición de la función 'Filter'

```
'
public void Filter(DelegateFilter filter)
{
    if(filter == null)
    return;

    for(int i = 0; i < list.Count; i++)
    {
        if(filter(list[i]))
            listResult.Add(list[i]);
    }
}
'
```

•Definición de la llamada a la función, mediante una función lambda.

```
'
Instancia i1 = new Instancia();
```

```
i1.Filter(node => { return node.Name == "hijo";});
'
```

•Podemos realizar código fuente utilizando patrones de programación, o respuestas establecidas para cada tipo de problema.
•Existen dos tipos de referencias a la hora de apuntar un objeto a la memoria RAM:
strong references: cada referencia aumenta el 'reference count'. *weak references*: las referencias no aumentan el 'reference count'.
•Los datos de tipo 'weak' no deberían ser los hijos, porque la ausencia de referencias hace que se elimine el objeto.

Clase 58

Jueves, 1 Febrero 2024

Indice

Continuación de la clase 'Tree'

- •Atributos:
  - •private Node<T> _parent
  - •private List<Node> _childen
  - •public T _item
- •Properties:
  - •IsRoot
  - •IsLeaf
  - •IsEmpty
  - •HasSiblings
  - •Level
  - •Root
  - •Parent
  - •ChildCount
- •Constructores:
- •Funciones:
  - •IndexOf()
  - •Unlink
  - •AddChildren
  - •SetParent
  - •AddListChildren
  - •ContainsChild
  - •ToString
- •Funciones 2:
  - •Filter
  - •Sort
  - •Visit

Clase 59

Viernes, 2 Febrero 2024

Indice

Continuacion de la clase 'Tree'

Clase 60

Lunes, 5 Febrero 2024

Indice

Significado de una Hash

•Un 'Hash' nos permite simplificar la definicion de un objeto. Por ejemplo, si un objeto es un número, el resto de una division seria el hash.
•Este numero Hash no debe repetirse para elementos diferentes. Cada objeto debe tener un hash propio, si no, no tendria sentido.
•Uno de sus usos mas empleados es para guardar la contraseña de los usuarios de un servidor.
•Si queremos comprobar si un objeto es igual a otro, en lugar de comprobar en profundidad sus atributos, podemos primero comprobar que tengan el mismo Hash. Solo en caso que tengan el mismo numero de referencia, comprobamos su semejanza en profundidad.
•C# contiene por defecto una clase llamada 'HashSet'. Su funcionamiento es el siguiente: comprueba el numero Hash y si coincide ejecuta el metodo 'Equals'
•Existe una norma que indica que si sobreescribes el Equals por defecto en una clase, es obligatorio sobreescribir el 'GetHashCode'

Clase 'SetWithHash'

•Crear una clase 'Set' que funcione con un 'Hash'

Clase 61

Martes, 6 Febrero 2024

Indice

Tipos de referencia.

•Para trabajar con Tree, tenemos que pasar las referencias a 'Strong' usando una property y trabajamos con esa variable
•De Weak Reference -> Strong Reference. Entonces usamos la Strong Reference.
•Es importante usar las properties lo menos posible.
•Cuando usamos dos variables, ponemos primero las mas cambiantes y en segundo lugar las variables menos cambiante.

Sentencias de control de flujo.

•Do...While
•Siempre se ejecuta el bucle **al menos una vez** antes de comprobar la condición de salida.
•Switch
•Try...Catch...Finally

Clase 'SetWithHash'

•Crear una clase 'Set' que funcione con un 'Hash'

Clase 70

Lunes, 15 Abril 2024

Indice

Introducción a las clases parciales

•Las clases parciales o 'partial class' son una característica propia del lenguaje C# y puede ser útil o no depende de como se use.
•Su funcionamiento es sencillo. Realizamos la definición de una clase de manera normal. Podemos establecer sus atributos, métodos y jerarquías de la manera que nos plazca. Entonces, si definimos la clase como **partial**, podemos crear *otra* clase, con el mismo nombre, que contenga funciones, métodos, atributos, etc **añadidos a la clase original.**
•Es decir, la definición de la clase se hace en secciones de programa distintas, siendo su comportamiento una combinación de las distintas partes.
•La sintaxis es como sigue:

```
'
public partial class Student
{
    // Definiciones
}
'
```

•Para que se comporte de manera parcial, debe tener **el mismo nombre de la clase.**
•Su uso, se establece según la práctica de cada despacho. Principalmente podemos establecer **dos** formas distintas de utilizar estas funcionalidades:
•1. Realizamos la definición de la clase según el uso que le queremos dar. Realizamos la definición completa. Cuando estamos completamente seguros que la clase funciona correctamente, su definición quedará guardada en un clase parcial, **y todas los cambios o implementaciones** de codigo nuevo que le queramos aportar a la clase, la realizamos en otra seccion. que sera 'partial' de la primera
•Es de alguna manera una copia de seguridad de un código que funciona, podemos pedirle a nuestro compañero que añada 'partial' a su sección y así nosotros podemos complementar la clase.
•2. Las clases partial se pensaron en su origen para comprimir secciones de código que quedarían ocultos a la vista del usuario.

•Así en una clase 'partial' añadimos el código que hace funcionar la clase pero no queremos que se vea, y en otra sección dejamos que el usuario complemente su sección de código
•Esto es lo que pasa con las 'partial class' de las aplicaciones de ventana **WPF.**

Introducción a las aplicaciones WPF

•Una aplicación de ventana WPF consta de dos partes diferenciadas que producen una experiencia de aplicación gráfica basada en C#
•Una parte donde se define la **apariencia visual** de la aplicacion o parte gráfica y otra parte que contiene el comportamiento de la aplicación.
•La primera se organiza con un código de tipo XML o XAML, donde se almacenan todas las variables gráficas. Se suele llamar ventana 'Designer'.
•La segunda se organiza con código fuente C#, que gestiona el funcionamiento del programa.
•La primera parte es todo código XAML que ocupa una clase parcial dentro del 'MainWindow'.
•La estructura jerárquica de cada uno de los elementos se almacena en lo que se denomina 'Arbol DOM' o 'Document Object Model'.
•En el apartado gráfico podemos añadir botones y objetos. Mediante los eventos dirigimos los objetos 'sender' hacia las funciones.
•Los eventos **siempre se designan añadiendo 'on' al nombre del elemento.**
•Por convención, se llama al elemento primero con el nombre de su función y luego con una nombre descriptivo. Por ejemplo, un botón puede recibir el nombre de 'ButtonMain', con el titulo de su función y el lugar que ocupa en el programa.

Clase 71

Martes, 16 Abril 2024

Indice

Introducción a las clases parciales

•Las clases parciales o 'partial class' son una característica propia del lenguaje C# y puede ser útil o no depende de como se use.
•Su funcionamiento es sencillo. Realizamos la definición de una clase de manera normal. Podemos establecer sus atributos, métodos y jerarquías de la manera que nos plazca. Entonces, si definimos la clase como **partial**, podemos crear *otra* clase, con el mismo nombre, que contenga funciones, métodos, atributos, etc **añadidos a la clase original.**
•Es decir, la definición de la clase se hace en secciones de programa distintas, siendo su comportamiento una combinación de las distintas partes.
•La sintaxis es como sigue:

```
'
public partial class Student
{
    // Definiciones
}
'
```

•Para que se comporte de manera parcial, debe tener **el mismo nombre de la clase.**
•Su uso, se establece según la práctica de cada despacho. Principalmente podemos establecer **dos** formas distintas de utilizar estas funcionalidades:
•1. Realizamos la definición de la clase según el uso que le queremos dar. Realizamos la definición completa. Cuando estamos completamente seguros que la clase funciona correctamente, su definición quedará guardada en un clase parcial, **y todas los cambios o implementaciones** de codigo nuevo que le queramos aportar a la clase, la realizamos en otra seccion. que sera 'partial' de la primera
•Es de alguna manera una copia de seguridad de un código que funciona, podemos pedirle a nuestro compañero que añada 'partial' a su sección y así nosotros podemos complementar la clase.
•2. Las clases partial se pensaron en su origen para comprimir secciones de código que quedarían ocultos a la vista del usuario.

•Así en una clase 'partial' añadimos el código que hace funcionar la clase pero no queremos que se vea, y en otra sección dejamos que el usuario complemente su sección de código
•Esto es lo que pasa con las 'partial class' de las aplicaciones de ventana **WPF.**