Tema 2 – Clases y objetos

¿Qué es una clase?

Una clase es una estructura de datos que se usa para definir tipos que extiendan los primitivos que provee el lenguaje. Las clases se utilizan por medio de sus **instancias**, las cuales se denominan objetos. Los objetos del mismo tipo (diferentes instancias de la misma clase) compartirán una estructura común, pero podrán poseer valores distintos en sus variables miembro.

Se podría trazar una analogía entre una clase y sus objetos con lo que es un molde y los elementos creados a partir de él. Una clase define una serie de variables y comportamientos por medio de los cuales se puede crear objetos que poseerán dichas variables y comportamientos.

Cómo declarar una clase

A diferencia de C++, en C# no hay un archivo de declaraciones (.h) y otro de definiciones (.cpp), sino que, por el contrario, todo el código que esté relacionado con una o más clases deberá ser incluido dentro de un mismo archivo (.cs). Lo más normal es colocar una clase por archivo .cs, pero eso es solo una recomendación, se puede incluir la cantidad de clases que se quiera.

```
class <identificador>{

}
```

En la declaración el identificador debe ser único en un namespace. El identificador de la clase debe tener un nombre que describa lo que representa la clase (Cliente, Producto, etc).

Modificadores de acceso

Dentro de lo que es el "cuerpo" de la clase se colocará la definición de todos sus miembros, que serán datos y métodos. Las variables miembro de una clase se declaran como variables convencionales, pero pueden poseer algunos modificadores de acceso, que regulan la visibilidad que posee la variable o el método.

Modificador	Acceso
public	Acceso desde cualquier ámbito
protected	Solo acceso desde la propia clase y clases derivadas.
private	Solo acceso desde la propia clase. **Por default, si no se especifica otra cosa, una variable es privada.**
internal	Solo acceso desde métodos de clases que se encuentren dentro del mismo **assembly**.
protected internal	Solo acceso desde métodos de clases que se encuentren dentro del mismo **assembly**, métodos de la misma clase y clases derivadas.

Instanciación

Para poder utilizar una clase hay que crear una instancia (aunque es posible acceder a variables y métodos estáticos sin la necesidad de crear objetos de la clase). Un objeto es una variable que lleva por tipo una clase.

```
Miclase miObjeto = new Miclase();
```

Un ejemplo de acceso a variables a través de una instancia.

```
class Miclase{
	public string var1 = "Hola";
	public string var2 = " como estas";
}
class MainClass{
	public static void Main (string[] args){
		Miclase miObjeto = new Miclase();
		Console.WriteLine("{0}{1}",miObjeto.var1, miObjeto.var2);

	}
}
```

Constantes de una clase

Las constantes dentro de una clase pueden poseer los mismo modificadores de acceso que las variables.

```
public const int var3 = 10;
```

Las constantes pueden ser accedidas sin requerir ninguna instanciación, ya que no tiene sentido almacenar en objetos valores que serán iguales para todos ellos sin excepción.

```
Console.WriteLine(Miclase.var3);
//Console.WriteLine(miObjeto.var3); // error de compilación
```

Variables estáticas

Pueden ser accedidas sin necesidad de instanciación, sin embargo, su valor puede ser modificado como cualquier otra variable. Estas variables se llaman variables de clase, y las tradicionales se las llama variables de objeto.

Este tipo de variables pueden poseer modificadores de acceso(public, private, protected...).

```
public static int var4 = 20;

Console.WriteLine(Miclase.var4);//20
Miclase.var4 = 200;
Console.WriteLine(Miclase.var4);//200
```

Los métodos

Las funciones declaradas dentro de una clase se conocen como métodos y contienen el código que será ejecutado por nuestro programa cuando se invocan.

Un método posee:

- Un tipo de dato: es el devuelto por el método cuando finaliza.
- Un identificador: nombre del método.
- Parámetros: lista de variables que recibe el método. Esta lista puede estar vacía.

Un ejemplo de uso de métodos con dos clases:

```
class Miclase{
	public string frase;

	public void imprimirFrase(){
		Console.WriteLine(this.frase);
	}
}
class MainClass{
	public static void Main (string[] args){
		Miclase miObjeto = new Miclase();
		miObjeto.frase = "cadena es lo mismo que decir string";
		miObjeto.imprimirFrase();

		Miclase miObjeto2 = new Miclase();
		miObjeto2.frase = "string es lo mismo que decir cadena";
		miObjeto2.imprimirFrase();
	}
}
```

Lo mismo pero con una sola clase y con variable y método private:

```
class Miclase{
	string frase;

	void imprimirFrase(){
		Console.WriteLine(this.frase);
	}

	public static void Main (string[] args){
		Miclase miObjeto = new Miclase();
		miObjeto.frase = "cadena es lo mismo que decir string";
		miObjeto.imprimirFrase();

		Miclase miObjeto2 = new Miclase();
		miObjeto2.frase = "string es lo mismo que decir cadena";
		miObjeto2.imprimirFrase();
	}
}
```

void indica que el método no devuelve ningún valor.

Métodos estáticos

Los métodos estáticos se llaman métodos de clase.

El método estático Main sirve como punto de entrada a la aplicación, contiene el código que se ejecutará primero en nuestro programa.

Este tipo de métodos no podrán acceder a variables de objeto(variables no estáticas) o métodos de objetos.

El modo de invocar un método estático es por medio del identificador de la clase en lugar del identificador del objeto.

```
class Miclase{
	public static void imprimirFecha(){
		Console.WriteLine("La fecha actual es {0}", DateTime.Now);
	}

	public static void Main (string[] args){
		Miclase.imprimirFecha();
	}
}
```

En el método estático imprimirFecha() se invoca al tipo estático Now de la clase DateTime. La librería BCL ofrece una estructura llamada DateTime que simplifica el manejo y unifica el almacenamiento de este tipo de datos.

Retornar valores

cuadradoValor() retorna un entero ya que se especifica el tipo de dato a retornar y al final del método se usa return que retorna el resultado del producto de la variable de instancia valor.

```
class Miclase{
	public int valor;
	public int cuadradoValor(){
		return this.valor * this.valor;
	}

	public static void Main (string[] args){
		Miclase miObjeto = new Miclase();
		miObjeto.valor = 10;
		Console.WriteLine("El valor del cuadrado de {0} es: {1}",
				miObjeto.valor, miObjeto.cuadradoValor());

	}
}
```

Uso de parámetros

Los parámetros sirven para enviar argumentos(valores o referencias) al invocar a un método. Estos argumentos serán usados por el método para realizar alguna acción. Normalmente todas las variables miembro son privadas y se accede a ellas por medio de métodos de tipo set y get para modificar o acceder a su valor.

```
class Miclase{
	private int valor;
	bool setValor(int num){
		//si el argumento esta en el rango 0-9
		if(num >= 0 && num < 10){
			this.valor = num;
			return true;
		}else
			return false;
		}

	int getValor(){
		return this.valor;
	}

	public static void Main (string[] args){
		Miclase miObjeto = new Miclase();
		if(miObjeto.setValor(55)){
			Console.WriteLine("El valor es {0}", miObjeto.getValor());
		}
		if(miObjeto.setValor(5)){
			Console.WriteLine("El valor es {0}", miObjeto.getValor());
		}

	}
}
```

Paso por valor

Usualmente en los métodos los parámetros de transfieren por copia de valores(a excepción del paso de objetos).

```
public static void setValor(int num){
	num = 10;
}

public static void Main (string[] args){
	int num = 20;
	Miclase.setValor(num);
	Console.WriteLine(num);//20
}
```

En num a la de la salida conserva su valor ya que lo que se pasa como argumento a setValor solo es una copia del valor de num pero no una referencia. Las dos variables se encuentran en posiciones de memoria distintas.

Paso por referencia

```
public static void setValorRef(ref int num){
	num = 10;
}

public static void Main (string[] args){
	int num = 20;
	Miclase.setValorRef(ref num);
	Console.WriteLine(num);//10
}
```

Una referencia es una variable que contiene la dirección de memoria de otra variable. En este caso se pasa al método una referencia a otra variable, por lo que lo que se haga con esta en el método también influirá a la variable externa al método.

Parámetros de salida

Si se necesita que un método devuelva más de un valor es posible especificar que ciertos parámetros sean de salida por medio del modificador **out**.

Ejemplo de método que devuelve el seno y coseno de un número pasado como argumento.

```
public static void raizRound(double dValor,
                             out double dRaiz, out double dRound){
      dRaiz = Math.Sqrt(dValor);
      dRound = Math.Round(dValor);
}

public static void Main (string[] args){
      double varRaiz, varRound;
      Miclase.raizRound(25.59, out varRaiz, out varRound);
      Console.WriteLine("Raiz: {0}", varRaiz);
      Console.WriteLine("Redondeo: {0}", varRound);
}
```

El método constructor

Existe un método especial que es invocado cada vez que un objeto nuevo es creado de forma automática. Este método se llama constructor y su identificador debe ser exactamente el mismo que el de la clase de la cual forma parte. El método constructor no posee tipo de datos de retorno. También es posible crear más de un constructor, cada uno con diferente cantidad de parámetros o con distintos tipos como parámetros.

```
class Vector{
      float x = 0.0f;
      float y = 0.0f;

      Vector(){
            Console.WriteLine("Metodo constructor sin parametros");
      }

      Vector(float x, float y){
            this.x = x;
            this.y = y;
            Console.WriteLine("Metodo constructor con dos parametros");
      }

      void mostrarValor(){
            Console.WriteLine("({0},{1})", this.x, this.y);
      }

      public static void Main (string[] args){
            Vector objeto = new Vector();
            objeto.mostrarValor();
            Vector objeto2 = new Vector(5.0f, 10.0f);
            objeto2.mostrarValor();
      }
}
```

Constructor de copia

Es posible especificar un constructor de copia para poder crear un objeto que sea igual a otro en contenido de datos.

```
Vector(Vector copia){
	this.x = copia.x;
	this.y = copia.y;
	Console.WriteLine("Metodo constructor de copia");
}

void mostrarValor(){
	Console.WriteLine("({0},{1})", this.x, this.y);
}

public static void Main (string[] args){
	Vector objeto2 = new Vector(5.0f, 10.0f);
	objeto2.mostrarValor();
	Vector objeto3 = new Vector(objeto2);
	objeto3.mostrarValor();
}
```

Es posible que un constructor invoque otro constructor. Esto suele realizarse cuando parte de la tarea de inicialización es común a todos los constructores; no tendría ningún sentido duplicar código.

Utilizar el operador de asignación no funciona para crear copias de objetos:

```
objeto3 = objeto2;
```

El objeto3 NO ES UNA COPIA DE OBJETO2, objeto3 es una REFERENCIA a objeto2. No se está copiando el contenido de un objeto en otro, solo se copia la referencia.

Destrucción de objetos

Los objetos poseen un ciclo de vida. Después de crearse, se hace uso de ellos y , llegado un punto, dejaran de ser útiles. Cuando un objeto deja de ser útil lo ideal es recuperar la memoria que ocupa.

C# gestiona la memoria de modo automático; para esto provee un recolector de basura (garbage collector) que se encarga de eliminar la memoria que en algún momento se solicitó y que ya no se utiliza más.

En C# no existe el operador contrario a new (como sí existe en C++, donde dicho operador se denomina delete). No se puede controlar exactamente el momento en el cual la memoria asignada vuelve a ser memoria disponible para otros objetos.

Lo que si se puede indicar al sistema cuándo un objeto deja de ser útil. Para esto bastará con pisar la referencia de un objeto por el valor null.

```
objeto3 = null;
```

Un objeto no poseerá todos los datos en su porción de memoria: las constantes y las variables estáticas, estarán ubicadas en un lugar único.

El destructor es método que se invoca cuando el objeto es destruido. Allí podremos colocar el código necesario para desinicializar el objeto. Su declaración es similar al constructor, sólo que posee el carácter ~ como prefijo.

```
~Vector(){
	Console.WriteLine("Destructor");
}
```

El destructor puede servir para cerrar un archivo y así asegurarse de que antes de que el garbage collector elimine el objeto, el archivo estaría cerrado. Este método solo es invocado por el garbage collector.

Pero no podemos estableces cuándo el garbage collector hará su trabajo. Hay una alternativa llamada Disposal que es un patrón de diseño, que consiste en crear un método específico que contenga todo el código de desinicialización. Este método podrá ser invocado manualmente.

```
public void Dispose(){
	Console.WriteLine("Dispose");
	GC.SuppressFinalize(this);
}
```

De esta manera el garbage collector eliminará el objeto sin invocar al destructor previamente.

Estructuras

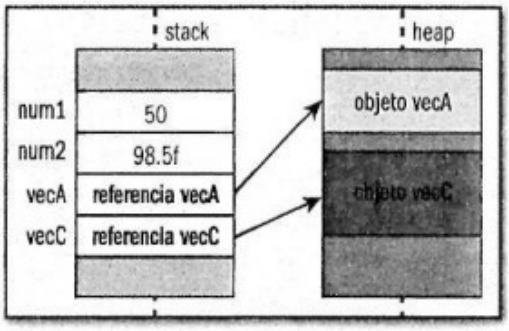
Una estructura es un tipo de dato muy similar a una clase. Puede contener prácticamente todos los elementos que puede poseer una clase. L**a diferencia principal que posee es que la estructura es un tipo de dato manejado por valor.**

```
namespace ejemplo_13
{
	struct Vector{
		float x;
		float y;

		public Vector(float x, float y){
			this.x = x;
			this.y = y;
		}

		public void mostrarValor(){
			Console.WriteLine("({0},{1})", this.x, this.y);
		}
	}

	class MainClass{
		public static void Main (string[] args){
			Vector vec = new Vector(5.0f, 3.0f);
			vec.mostrarValor();
		}
	}
}
```

Las diferencias entre una estructura y una clase son:

- No es válido inicializar las variables en la misma línea que su declaración.
- No es posible especificar constructores sin parámetros.
- No es posible especificar destructores.
- No es posible practicar herencia. Las estructuras son derivadas de la clase Object.

Las estructuras son asignadas en el stack, mientras que, por el contrario, los objetos son asignados en el heap.



Para poder utilizar una estructura, es necesario previamente asignar valor a todas sus propiedades. Es posible el uso de **new** con las estructuras si se desea invocar algún constructor en particular. Es importante saber si dicho tipo es una clase o una estructura para no cometer errores.

REPASO A CONCEPTOS SOBRE LA MEMORIA

La memoria está dividida en tres partes, Zona de Datos, Stack y Heap.

Zona de Datos

La Zona de Datos es donde se guardan las instrucciones del programa, las clases, los métodos, y las constantes. Esta parte de la memoria es totalmente fija, y nada durante el tiempo ejecución lo puede cambiar.

Stack

El Stack, para explicarlo de una manera más rápida, se puede imaginar como si fuese un Array, que **es de tamaño fijo durante el tiempo ejecución del programa, en tiempo de compilación es cuando se define el tamaño que tendrá**. En concreto **se guardan dos tipos de elementos, las referencias a objetos(o instancias) y datos de tipo primitivo**(int, float, char…). También se almacenan las variables locales, parámetros/valores de retorno de los métodos, etc.

Por ejemplo, si ejecutamos un método, las variables locales de ese método se guardarían en el Stack, y al finalizar el método, se borran del Stack, pero hay que tener en cuenta que el Stack es fijo, y solamente estaríamos usando y dejando de usar una parte del Stack, y por esa misma característica de ser estático, si lo llenamos, caeríamos en un StackOverflowError.

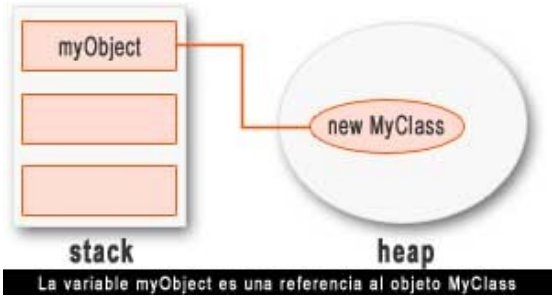
Heap

El Heap es la **zona de la memoria dinámica, los objetos son creados, eliminados o modificados en esta parte de la memoria**.

Como interactúan el Heap y el Stack

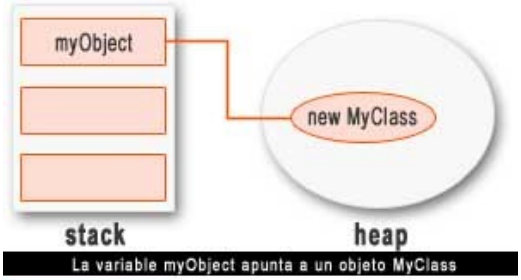
El Stack guarda referencias y datos primitivos, las referencias pueden apuntar a elementos del heap(es decir, a los objetos). Las referencias tienen un tipo que está determinado por la clase de la que se creará la instancia, y puede apuntar a esa clase o subclases(elementos que hereden de la clase).

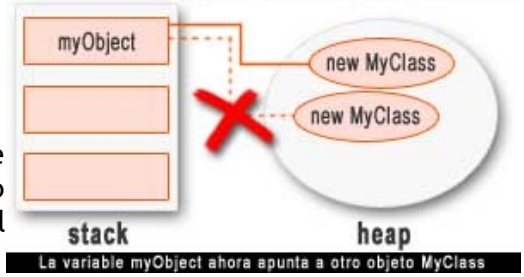
```
class MyClass{
	public static void Main (string[] args){
		MyClass myObject; // se crea la referencia en el stack
		myObject = new MyClass();// se crea el objeto en el heap
		// con referencia al stack
	}
}
```




```
class MyClass{
	public static void Main (string[] args){
		MyClass myObject;
		myObject = new MyClass();
		myObject = new MyClass();
	}
}
```

Primero se crea la referencia, luego el objeto, y se hace que la referencia apunte hacia él, luego se crea otro objeto y se hace que la referencia ahora apunte hacia el nuevo objeto.





La relación referencia-objeto es de muchos a uno, es decir, un objeto puede estar siendo apuntado por muchas referencias, pero una referencia solo apunta a un objeto.

Relación con el GC Garbage Collector

Si los objetos están perdidos en memoria, no tienen referencias, y por lo tanto no se pueden acceder, esos son los candidatos a ser borrados por el GC.

```
class MyClass{
	public static void Main (string[] args){
		MyClass myFirstObject = new MyClass();//3
		MyClass mySecondObject = new MyClass();//4
		myFirstObject = mySecondObject;//5
		mySecondObject = null;//6
	}
}
```

Línea 3. Se crea el primer objeto en el heap, se crea la primera referencia en el stack, y la referencia apunta al objeto.

Línea 4. Se crea el segundo objeto en el heap, se crea la segunda referencia en el stack, y la referencia apunta al objeto.

Línea 5. La primera referencia apunta hacia el segundo objeto. En este momento el primer objeto se queda sin referencia, es desechado por el GC, y se libera memoria.

Línea 6. La segunda referencia, apunta a nada, por lo que también el segundo objeto es desechado por el GC, y se libera memoria.

ACTIVIDADES:

1.- ¿Donde se encuentran los errores?. Corregidlos.

a)

```
class MyClass{
	const int a;
	int b;
	int c;
}
```

b)

```
class MyClass{
	public int m_iValor;
	static void imprimirValor(){
		Console.WriteLine(m_iValor);
	}
}
```

c)

```
class MyClass{
	public int m_iValor;
	bool fijarValor(int iValor){
		if(iValor >= 0 && iValor < 10){
			m_iValor = iValor;
			return true;
		}
	}
}
```

2.-¿Es necesario instanciar una clase para acceder a: (marcar correcta)

a) Una variable de tipo entero no estática?
b) Una variable de tipo entero estática?
c) Un método estático?

3.-¿Cuáles son las diferencias principales entre clases y estructuras?

4.-¿Pueden las estructuras poseer constructores? ¿En qué forma?

5.-¿Cuál es el nivel de acceso que se usa por defecto en miembros de clase sino se especifica lo contrario?