Universidad del Istmo de Guatemala
Facultad de Ingenieria
Ing. en Sistemas
Informatica 2
Prof. Ernesto Rodriguez - erodriguez@unis.edu.gt

Hoja de trabajo #3

Fecha de entrega: 15 de Febrero, 2018 - 11:59pm

Instrucciones: Realizar cada uno de los ejercicios siguiendo sus respectivas instrucciones. El trabajo debe ser entregado a traves de Github, en su repositorio del curso, colocado en una carpeta llamada "Hoja de trabajo 3". Al menos que la pregunta indique diferente, todas las respuestas a preguntas escritas deben presentarse en un documento formato pdf, el cual haya sido generado mediante Latex. Los ejercicios de programación deben ser colocados en una carpeta llamada "Programas", la cual debe colocarse dentro de la carpeta correspondiente a esta hoja de trabajo.

Parametros de retorno

En C# hay varias opciones para indicar que un metodo no se pudo ejecutar correctametne debido a que los parametros que se le dieron eran invalidos. Una forma de hacer esto es utilizar *parametros de retorno*.

Los parametros de retorno se definen colocando la palabra reservada `out` antes del tipo de un parametro en la firma del metodo. Ellos permiten que el codigo que invoca al metodo pueda pasarle variables al metodo las cuales pueden utilizarse para almacenar el valor retornado. Considere el siguiente codigo:

```
public class Program{
    public static bool Dividir(double num, double den, out double resultado){
        if(den != 0){
            resultado = num / den;
            return true;
        }else{
            return false;
        }
    }

    public static void Main(string[] args){
        double resultado = 0;

        if(Program.Dividir(32, 0, resultado)){
            Console.WriteLine($"La respuesta es {resultado}");
        }else{
            Console.WriteLine($"Error, el divisor no puede ser '0'!");
        }
    }
}
```

Este programa tiene un metodo `Dividir`, el cual verifica que el divisor sea diferente de cero antes de hacer la division. Cuando el divisor es diferente de cero, el resultado se almacena en la variable `resultado` y luego se retorna `true` para indicar que la division fue exitosa. De lo contrario, el metodo solamente retorna `false`.

Cuando este metodo es llamado, debe recibir una variable como tercer parametro. Al hacer eso, el metodo `Dividir` obtiene acesso a la variable *respuesta* donde escribe su resultado en caso que la division sea exitosa. Luego el metodo `main` puede verificar si la division fue exitosa mediante un `if`, e imprimir la respuesta en caso exitoso o mostrar el error en caso contrario.

En el caso de este ejemplo, el programa imprimiria en la consola "Error, el divisor no puede ser '0'!" debido a que el divisor que se utiliza es '0';

Iniciacion

1. Adentro de la carpeta programas, crear una solución con el mismo nombre, mediante `dotnet new sln`
2. Crear dos carpetas dentro de la carpeta "Progrmas" llamadas "List" y "ListTests"
3. Crear un proyecto de tipo *consola* en la carpeta "List" mediante `dotnet new console`
4. Crear un proyecto de tipo *xunit* dentro de la carpeta "ListTests" mediatne el comando `dotnet new xunit`

Ejercicio #1 (25%)

En el proyecto "List", defina una interfaz llamada `IList` y coloquela en un archivo llamado "IList.cs". Esta interfaz representa una lista generica de objetos arbitrarios, por lo cual debe tener un parametro generico llamado `T`. La interfaz debe definir los siguientes metodos:

Nombre	**Tipo de parametros**	**Tipo de retorno**	**Descripción**
`Get`	`int` $\otimes$ `out T`	`bool`	Obtener objeto por indice.
`Set`	`int` $\otimes$ `T`	`bool`	Colocar valor en el indice.
`Push`	`T`	`void`	Colocar elemento al final de la lista incrementando su longitud por 1
`Length`		`int`	Obtener la longitud de la lista

Nota: la notación `int` $\otimes$ `string` *es una notación comun en el campo de la computación para denotar firmas de metodos. En este caso, la notacion indica que el metodo acepta un* `int` *como primer parametro y un* `string` *com segundo parametro.*

Recordar que las interfaces solo definen la *firma* de los metodos, no los cuerpos.

Ejercicio #2 (25%)

En el proyecto "List", definir una clase abstracta llamada `Lista`. Esta clase abstracta debe implementar la interfaz `IList` declarando los metodos definidos en dicha interfaz como *metodos abstractos*. Para ello, esta clase también debe aceptar un parametro generico `T`. Adicionalmete, la clase abstracta debe definir un metodo concreto `Push` : `IList`$\langle$`T`$\rangle \Rightarrow$ `void` (recibe como parametro un `IList` del mismo tipo que la lista y

retorna void), el cual tiene el mismo nombre al metodo `Push` definido anteriormete, que es un *overload* del metodo `Push` el cual empuja todos los elementos de la lista que se paso como parametro a la lista actual.

Ejercicio #3 (25%)

En el proyecto "List", definir una clase llamada `ArrayList` la cual hereda de la clase abstracta *List*. El constructor de esta clase recibe un arreglo de `T` (el parametro generico) y utiliza ese arreglo como su almacenamiento inicial. Esto significa que la clase debe tener una propiedad privada llamada `almacenamiento` de tipo `T[]` la cual se inicializa con el arreglo obtenido como parametro. Cada vez que se haga una llamada al metodo `Push`, la clase debe:

1. Crear un arreglo nuevo de longitud `almacenamiento.Length` + 1
2. Copiar todos los elementos existentes en el mismo orden a este nuevo arreglo.
3. Colocar el elemento que se paso como parametro e la ultima celda del arreglo.
4. Asignar este arreglo a la propiedad `almacenamiento` de la clase.

Los métodos `Get`, `Set` y `Length` de esta clase deben utilizar el arreglo `almacenamiento` para llevar a cabo su trabajo.

Ejercicio #4 (25%)

En el proyecto "ListTests", escribir tests unitarios que validen lo siguiente:

1. Al llamar el metodo `Push` de un `ArrayList`, su longitud aumenta en 1 y su ultimo elemento (indice longitud menos 1) es el elemento que fue empujado.
2. Al llamar el metodo `Get`, con un indice mayor a la longitud de la lista, el metodo retorna `false`.
3. Crear un test unitario propio. Recordar que los tests unitarios deben ir en un proyecto diferente al proyecto donde se definieron las clases anteriores.