La Programación Básica en Entorno Visual (I)

UNIDAD N° 2

(Parte 3/3)

2023

Temas

- Trabajando con variables, operadores y expresiones. Escribir métodos y definir ámbitos.
- Usando operadores de asignación compuestos y declaraciones de iteración.
- Manejando errores y excepciones.
- Repasando: tipos por valor y referencia, arreglos.
- Creando y manejando clases y objetos. Trabajando con herencia.
- Implementando propiedades para el acceso a campos.
- Creando tipos con enumeraciones y estructuras.
- Usando arrays. Entendiendo los parámetros de arrays.
- Trabajando con interfaces y definiendo clases abstractas.
- Usando y enumerando colecciones. Consultando datos en memoria con lenguaje de consulta.

CHAPTER 10

Using arrays

Populating and using an array

```
int[] pins = new int[4]{ 9, 3, 7, 2 };
```

```
int[] pins = new int[3]{ 9, 3, 7, 2 }; // compile-time error
int[] pins = new int[4]{ 9, 3, 7 };    // compile-time error
int[] pins = new int[4]{ 9, 3, 7, 2 }; // OK
```

Creating an implicitly typed array

```
var names = new[]{"John", "Diana", "James", "Francesca"};
```

```
var names = new[] { new { Name = "John", Age = 47 },
                    new { Name = "Diana", Age = 46 },
                    new { Name = "James", Age = 20 },
                    new { Name = "Francesca", Age = 18 } };
```

Copying arrays

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins; //  alias and pins refer to the same array instance
```

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
for (int i = 0; i < pins.Length; i++)
{
    copy[i] = pins[i];
}
```

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0);
```

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```

CHAPTER 11

Understanding parameter arrays

There are several points worth noting about *params* arrays:

- You can't use the *params* keyword with multidimensional arrays. The code in the following example will not compile:

```
// compile-time error
public static int Min(params int[,] table)
...
```

- You can't overload a method based solely on the *params* keyword. The *params* keyword does not form part of a method's signature, as shown in this example:

```
// compile-time error: duplicate declaration
public static int Min(int[] paramList)
...
public static int Min(params int[] paramList)
...
```

- You're not allowed to specify the *ref* or *out* modifier with *params* arrays, as shown in this example:

```
// compile-time errors
public static int Min(ref params int[] paramList)
...
public static int Min(out params int[] paramList)
...
```

- A *params* array must be the last parameter. (This means that you can have only one *params* array per method.) Consider this example:

```
// compile-time error
public static int Min(params int[] paramList, int i)
...
```

- A non-*params* method always takes priority over a *params* method. This means that if you want to, you can still create an overloaded version of a method for the common cases, such as in the following example:

```
public static int Min(int leftHandSide, int rightHandSide)
...
public static int Min(params int[] paramList)
...
```

Using *params object[ ]*

```
class Black
{
    public static void Hole(params object [] paramList)
    ...
}
```

- You can pass the method no arguments at all, in which case the compiler will pass an object array whose length is 0:

```
Black.Hole();
// converted to Black.Hole(new object[0]);
```

Using *params object[ ]*

- You can call the *Black.Hole* method by passing *null* as the argument. An array is a reference type, so you're allowed to initialize an array with *null*:

```
Black.Hole(null);
```

- You can pass the *Black.Hole* method an actual array. In other words, you can manually create the array normally generated by the compiler:

```
object[] array = new object[2];
array[0] = "forty two";
array[1] = 42;
Black.Hole(array);
```

- You can pass the *Black.Hole* method arguments of different types, and these arguments will automatically be wrapped inside an *object* array:

```
Black.Hole("forty two", 42);
//converted to Black.Hole(new object[]{"forty two", 42});
```

CHAPTER 13

Creating interfaces and defining abstract classes

Abstracciones

Normalmente, las clases se utilizarán para modelar abstracciones extraídas del problema que se intenta resolver o de la tecnología que se utiliza para implementar una solución a ese problema. Cada una de estas abstracciones es parte del vocabulario del sistema. Las clases bien estructuradas:

- Proporcionan una abstracción precisa de algo extraído del vocabulario del problema o del dominio de la solución.
- Contiene un pequeño conjunto bien definido de responsabilidades y las lleva a cabo todas ellas muy bien.
- Proporciona una clara distinción entre la especificación de la abstracción y su implementación.
- Es comprensible y sencilla, a la vez que extensible y adaptable.

Separación de Intereses (*Separation of Concerns*)

En software es importante construir sistemas con una clara separación de intereses, de forma que, al evolucionar el sistema, los cambios en una parte del sistema no se propaguen, afectando a otras partes del sistema.

Una forma importante de lograr este grado de separación es especificar unas líneas de separación claras en el sistema, estableciendo una frontera entre aquellas partes que pueden cambiar independientemente.

Definición

Las interfaces definen una línea entre la especificación de lo que una abstracción hace y la implementación de cómo lo hace. Una interfaz es una colección de operaciones que sirven para especificar un servicio de una clase o de un componente.

Ejemplo

Interface restrictions

- You're not allowed to define any fields in an interface, not even static fields. A field is an implementation detail of a class or structure.
- You're not allowed to define any constructors in an interface. A constructor is also considered to be an implementation detail of a class or structure.
- You're not allowed to define a destructor in an interface. A destructor contains the statements used to destroy an object instance. (Destructors are described in Chapter 14, "Using garbage collection and resource management.")
- You cannot specify an access modifier for any method. All methods in an interface are implicitly public.
- You cannot nest any types (such as enumerations, structures, classes, or interfaces) inside an interface.
- An interface is not allowed to inherit from a structure or a class, although an interface can inherit from another interface. Structures and classes contain implementation; if an interface were allowed to inherit from either, it would be inheriting some implementation.

CHAPTER 17

Introducing generics

Ejemplo

- **Covariance example** If the methods in a generic interface can return strings, they can also return objects. (All strings are objects.)
- **Contravariance example** If the methods in a generic interface can take object parameters, they can take string parameters. (If you can perform an operation by using an object, you can perform the same operation by using a string because all strings are objects.)

CHAPTER 18

Using collections

Collection	Description
List<T>	A list of objects that can be accessed by index, like an array, but with additional methods to search the list and sort the contents of the list.
Queue<T>	A first-in, first-out data structure, with methods to add an item to one end of the queue, remove an item from the other end, and examine an item without removing it.
Stack<T>	A first-in, last-out data structure with methods to push an item onto the top of the stack, pop an item from the top of the stack, and examine the item at the top of the stack without removing it.
LinkedList<T>	A double-ended ordered list, optimized to support insertion and removal at either end. This collection can act like a queue or a stack, but it also supports random access like a list.
HashSet<T>	An unordered set of values that is optimized for fast retrieval of data. It provides set-oriented methods for determining whether the items it holds are a subset of those in another *HashSet<T>* object as well as computing the intersection and union of *HashSet<T>* objects.
Dictionary<TKey, TValue>	A collection of values that can be identified and retrieved by using keys rather than indexes.
SortedList<TKey, TValue>	A sorted list of key/value pairs. The keys must implement the *IComparable<T>* interface.

One other important set of collections is available, and these classes are defined in the *System.Collections.Generic.Concurrent* namespace. These are thread-safe collection classes that you can use when building multithreaded applications. Chapter 24, "Improving response time by performing asynchronous operations," provides more information on these classes.

Note As with arrays, if you use *foreach* to iterate through a *List<T>* collection, you cannot use the iteration variable to modify the contents of the collection. Additionally, you cannot call the *Remove*, *Add*, or *Insert* method in a *foreach* loop that iterates through a *List<T>* collection; any attempt to do so results in an *InvalidOperationException* exception.

Comparing arrays and collections

Here's a summary of the important differences between arrays and collections:

- An array instance has a fixed size and cannot grow or shrink. A collection can dynamically resize itself as required.
- An array can have more than one dimension. A collection is linear. However, the items in a collection can be collections themselves, so you can imitate a multidimensional array as a collection of collections.
- You store and retrieve an item in an array by using an index. Not all collections support this notion. For example, to store an item in a *List<T>* collection, you use the *Add* or *Insert* methods, and to retrieve an item, you use the *Find* method.
- Many of the collection classes provide a *ToArray* method that creates and populates an array containing the items in the collection. The items are copied to the array and are not removed from the collection. Additionally, these collections provide constructors that can populate a collection directly from an array.

CHAPTER 19

Enumerating collections

Enumerating the elements in a collection

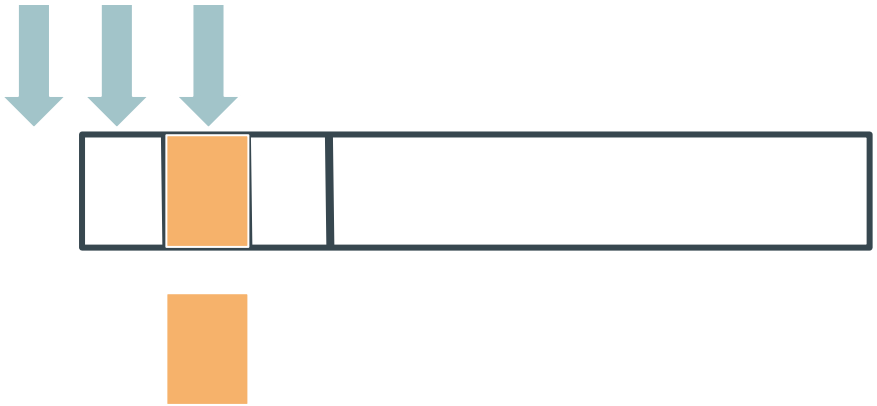
```
List<int> precios = new List(){10,20,100,50};
foreach (int precio in precios)
{
    Console.WriteLine(precio);
}
```

***enumerable* collection**

System.Collections.IEnumerable

IEnumerator

```
bool MoveNext();
object Current {get;set;}
void Reset();
```



Implementing an enumerator by using an iterator