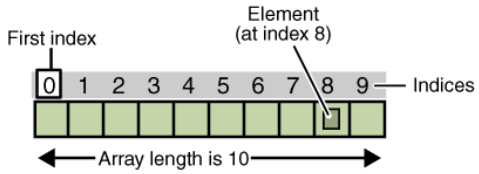
Arrays

Un array es una lista ordenada de elementos. Pueden ser números, cadenas de texto o cualquier otro tipo de objeto.



Definición

En C# todos los elementos de un array tienen que ser del mismo tipo. Es necesario declarar de que tipo van a ser los elementos de un array al definirlo:

```
int[] listaDeNumeros;
```

Como veis, para declarar un array en C# se pone el tipo que tendrán sus elementos seguido de corchetes (apertura, `[`, y cierre `]` sin dejar espacio entre medias) y después el nombre de la variable.

Cuando los utilices en tus programas, tendrás que decidir si los arrays que declares son locales o propiedades de la clase (con `public` o `private` delante y definidos al principio de la clase y fuera de cualquier función)

Pero todavía no hemos *creado* ningún array, sólo hemos definido una variable de tipo *array de enteros*. Su valor demomento y hasta que le asignemos un array será por defecto `null`.

Asignación

Para asignar un nuevo array a una variable de este tipo hay que crear un nuevo array o llamar una función que devuelva un array.

Como crear un array

Para crear un array debemos utilizar la palabra clave `new`, seguido del tipo de los elementos que contendrá el array y del número de elementos que queramos que tenga entre corchetes:

```
listaDeNumeros = new int[5];
```

En el ejemplo anterior estamos asignando a la variable `listaDeNumeros` nuevo un array de enteros con 5 elementos.

Con esta sentencia no hemos dicho cuales son esos elementos, es decir, no hemos **inicializado** el array. Como un número entero por defecto es 0, tendremos un array con 5 ceros.

Cuidado: Qué valores tienen por defecto los elementos de un array que no ha sido inicializado depende del tipo que hayamos declarado para sus valores. Si fuera un array de `string`s

```
string[] listaDePalabras = new string[5];
```

el array contendría 5 valores `null`.

Para asignar otros valores a los elementos del array tendremos que ir uno a uno cambiando su valor. Sólo se puede cambiar el valor de sus elementos usando la notación con corchetes que ya hemos visto:

```
listaDeNumeros[0] = 2;
listaDeNumeros[1] = 4;
listaDeNumeros[2] = 6;
listaDeNumeros[3] = 8;
listaDeNumeros[4] = 10;
```

Ahora, el array `listaDeNumeros`, contiene los cinco primeros números pares.

Si el array es muy largo y podemos crear una fórmula para sus elementos, es más cómodo utilizar un bucle. En C# los bucles `for` se escriben prácticamente igual que en cualquier otro lenguaje, sólo que hay que decir explícitamente que la variable que hace de contador es de tipo int:

```
int counter = 0;
while (counter < 5) {
    listaDeNumeros[counter] = counter * 2;
    counter += 1;
}
```

Este búcle hace lo mismo que el ejemplo anterior, pero en menos líneas. Fíjate bien en la sentencia de dentro. Procura entender bien lo que está pasando.

Expresiones de inicialización

Para simplificar la creación de nuevos a rrays C# tiene lo que se llama expresión de inicialización para arrays.

Siguiendo con el ejemplo de los números, podríamos rellenar el array en el momento en el que lo definimos de esta forma:

```
int[] listaDeNumeros = new string[5] {2, 4, 6, 8, 10};
```

O incluso así, que es más corto:

```
int[] listaDeNumeros = {2, 4, 6, 8, 10};
```

Propiedades y métodos útiles de la clase Array

Todos los arrays que crees con las técnicas del apartado anterior son *instancias* u *objetos* de la clase `Array` (que está definida en el espacio de nombres `System`)

Eso significa que todos los arrays tienen las propiedades y métodos definidos en dicha clase. Estos son los más útiles:

- **`Length`**: contiene la longitud del array, el número de elementos que contiene.
- **`IndexOf(elemento)`**: devuelve el índice que tiene un determinado `elemento` dentro del array. Si el elemento no está en el array, devuelve -1.
- **`Clone()`**: devuelve una copia del array. Ejemplo:
  ```
  int[] copiaListaDeNumeros = (int[]) listaDeNumeros.Clone();
  ```
  Es necesario convertir el valor develto por clone porque es un objeto genérico.

Fíjate en que no tenemos funciones como `push()` o `pop()`como el año pasado. Esto es porque **los arrays en C# son de longitud fija y no se pueden añadir elementos ni eliminar elementos**.

Cuando necesitemos una lista en la que se puedan añadir o eliminar elementos utilizaremos otros tipo de datos, que también son colecciones de elementos, y que veremos más adelante.

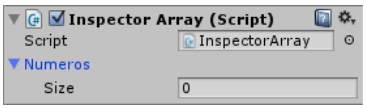
Otra forma de crear e inicializar arrays

En Unity, podemos definir el array como una propiedad pública de la clase.

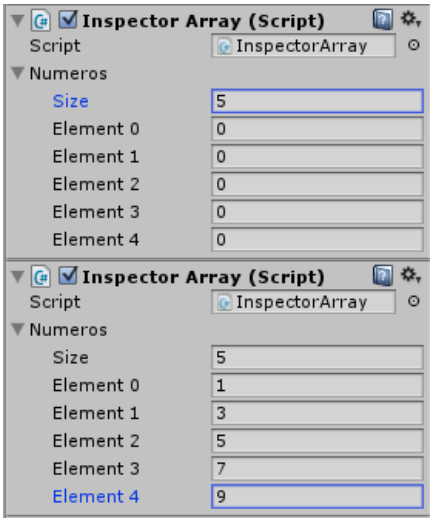
```
//...

    [SerializeField] private int[] numeros;
//...
```

Al hacer esto aparecerá en el inspector:

Si expandimos la propiedad numeros:



Podremos introducir el número de elementos que deseamos, y rellenar en el inspector cada uno de los valores de sus elementos:



De este modo **delegamos en el inspector** la creación e inicialización del array.

Inicializar arrays en el inspector de longitud prefijada en el código (evítalo)

Si es lo que queremos, podemos delegar en el inspector sólo la inicialización del array, creandolo nosotros en el código con la longitud que deseemos:

```
//...

    public int[] numeros =  new int[5];
//...
```

De esta forma el array `numeros` tendrá una longitud inicial de 5 elementos, y ya aparecerán creados en el inspector.

El problema que tiene esto es que es confuso, porque si cambiamos *Size en el inspector* cambiamos la longitud del array y podemos introducir más elementos.

Sin embargo nuestro código no cambia. Cuando ejecutemos el programa la longitud real que tendrá el array será la que hay en el inspector, pero como lo que nosotros vemos cuando escribímos es el código, es muy fácil olvidarse de esto y creer que la longitud del array es 5 cuando en el inspector podríamos haber introducido 127, por decir algo.

Los arrays pueden tener varias dimensiones.

Con dos tenemos bastante. ¿Cómo es un array de dos dimensiones? Hemos visto que los arrays pueden ser de enteros, de cadenas de texto, de gameObjects, de Transforms, ... y tambien podemos crear arrays de arrays.

Eso es un array de dos dimensones, un array de arrays. Se escribe así:

```
string[,] colors= new string[20,20];
```

Esta sentencia crearía un array de 20x20 colores que podría servir para configurar una rejilla bidimensional a dibujar en la pantalla.

Más en el contexto de Unity, podríamos utilizar un array bidimensional para configurar una rejilla de gameObjects que han de colocarse de forma ordenada en una superficie rectangular.

```
GameObject[,] prefabMap = new GameObject[50,50];
```

Los arrays bidimensionales se utilizan muy a menudo para definir los mapas de los niveles en los juegos.

Puedes leer más detalles y ver ejemplos de arrays bidimensionales (para C# en general no específicamente para Unity) aquí

Los arrays bidimensionales no se pueden inicializar en el inspector. Es decir, que hay que hacerlo en el código, con bucles o con la sintáxis de llaves, como vimos en el apartado anterior:

```
public int[,] numeros2D = new int[3, 2] {{1, 5 }, {4, 215}, {23, 17}};
```

Como ves los arrays bidimensionales no tienen porque ser *"cuadrados"*.
Lo anterior se puede separar en varias líneas para que resulte más fácil de leer y también de rellenar

```
public int[,] numeros2D = new int[3, 2] { { 1, 5 }, { 4, 215}, { 23, 17 } };
```

Para leer o modificar los valores de los arrays de dos dimensiones se suelen utilizar dos bucles, uno anidado dentro del otro.

```
for (int i = 0; i< 3; i++) {
    for (int j = 0; j < 2; j++) {
        print(numeros2D[i,j]);
    }
}
```

La longitud de un array en 2 dimensiones

La propiedad `Length` en un array de dos dimensiones deja de tener sentido, porque lo que normalmente queremos saber es la longitud del array en una de las dos dimensiones, por ejemplo para decidir cuantas veces ha derepetirse cada uno de los bucles anidados que usábamos en el ejemplo anterior para mostrar los elementos del array.

En este caso necesitamos la función `getLength()` de los arrays. ¿Por qué una función? Porque necesitamos especificar de cual de las dos dimensiones queremos la longitud. Si queremos saber el numero de filas (la primera longitud que especificamos al crear el array, 3 en este caso) usaremos `getLength(0)`. Si queremos saber el número de columnas (la segunda dimensión, que tiene logitud 2 en nuestro ejemplo) usaremos `getLength(1)`.

Por tanto, la función `getLength()` devuelve el número de elementos en la dimensión que especifiquemos como parámetro: 0 para la primera (filas), 1 para la segunda (columnas).

Podemos entonces reescribir el ejemplo anterior de esta forma:

```
public int[,] numeros2D = new int[3, 2] {
{ 1, 5 },
{ 4, 215},
{ 23, 17 }
};

void Start() {
    for (int i = 0; i < numeros2D.getLength(0); i++) {
        for (int j = 0; j < numeros2D.getLength(1); j++) {
            print(numeros2D[i,j]);
        }
    }
}
```

Segútamente te estarás preguntando para que queremos usar `getLength()` si ya sabemos que el número de filas es 3 y el número de columnas es 2. Muy fácil, si cambiamos el tamaño del array `numeros2D`, no tenemos que cambiar el límite de los arrays.

Es más, el array `numeros2D` podría haber sido creado de forma automática en alguna otra parte de mi código, de forma dinámica y dependiendo de lo que haya ocurrido en mi juego, de forma que no puedo saber su tamaño cuando escribo el código, ya que eso se decide mientras se juega.

List<T>

Al principio hay tres elementos, pero según avanza el juego hay 9, y luego 12, más tarde 32, y luego van disminuyendo hasta llegar a cero... Pueden ser enemigos, trampas, objetos que necesitamos recoger en el nivel actual... Hay muchas situaciones en las que necesitamos un número **variable** de elementos del mismo tipo y necesitamos tenerlos en algún tipo de lista para poder recorrerlos y hacer operaciones sobre ellos, o simplemente contarlos.

Definir listas es tan fácil como definir arrays (aunque la sintaxis es diferente):

```
List<GameObject> enemies;
```

Para crear una lista y asignársela a la variable anterior, escribiríamos:

```
enemies = new List<GameObject>();
```

Ésta es la versión más sencilla de crear una lista vacía. Después podemos ir añadiendo elementos usando su método `Add()` (ver siguiente sección). A veces nos interesa indicar una longitud inicial o rellenar la lista nada más empezar.

Para indicar el número de elementos podríamos hacer:

```
enemies = new List<GameObject>(10);
```

Pero esto no sirve para lo que podríamos creer. En esto las listas son radicalmente diferentes a los arrays. La línea anterior sólo reserva memoria para poder añadir 10 elementos en la lista (con Add()) sin que la máquina cirtual tenga que reservar memoria adicional cada vez que se añada un elemento. Esto puede ser útil en una aplicación para móvil, por ejemplo, pero no crea los elementos de la lista. `enemies.Count` seguirá siendo 0 después de esto.

Lo que sí interesa a veces es rellenar la lista nada más empezar:

```
enemies = new List<GameObject>() {
Instantiate(dumbEnemyPrefab),
  Instantiate(smartEnemyPrefab),
  Instantiate(crazyEnemyPrefab),
  Instantiate(dumbEnemyPrefab),
  Instantiate(dumbEnemyPrefab),
  Instantiate(bossEnemyPrefab)
};
```

Pero lo más habitual es rellenar la lista con un bucle:

```
enemies = new List<GameObject>();
int counter = 0
while (counter < 10) {
  enemies.Add(Object.Instantiate(dumbEnemyPrefab));
  counter += 1;
}
```

A veces, tenemos los elementos que nos interesa en un array, pero necesitamos poder quitarle elementos o añadírselos. Esta sería una forma de conver el array `enemiesArr` en una lista:

```
List<GameObject> enemiesList = new List<GameObject>(enemiesArr);
```

Las listas requieren usar un nuevo espacio de nombres

Es posible que mientras leías los ejemplos anteriores hayas intentado probarlos en un proyecto de Unity. Te habrás dado cuenta de que no funciona. Lo más probable es que hayas encontrado con el siguiente error:

```
error CS0246: The type or namespace name `List` could not be found. Are you missing a using directive or
an assembly reference?
```

¿Recuerdas esas dos líneas que aparecen al principio de nuestras clases y que nunca has entendido para que sirven? Pues ahora necesitamos hacer cambios en esa parte de nuestro código.

```
using UnityEngine;
using System.Collections;
```

Estas sentencias sirven para que el compilador pueda encontrar las definiciones de las clases de Unity que necesitamos para nuestros juegos las de las colecciones de datos por defecto de C# (que por cierto no hemos usado nunca, los arrays no están en ese paquete). `UnityEngine` y `System.Collections` son espacios de nombres. Son agrupaciones de clases que se pueden incluir en nuestro programa de forma separada. Hay muchas clases predefinidas tanto en C# como en el motor de Unity. Si se incluyeran todas por defecto, nuestro juego compilado podría ocupar gigas (bueno, quizás aquí este exagerando un poco, pero entiendes a lo que me refiero ¿no?). Mediante este mecanismo, podemos incluir sólo los espacios de nombres que necesitamos para nuestro programa.

Así, por ejemplo, como nunca hemos utilizado las colecciones de datos estándar de C# podríamos borrar de todos nuestros programas la línea **using** System.Collections , pero no te preocupes, tampoco está tan mal dejarla ahí, sólo estamos desperdiciando unos pocos Kb en este caso.

Lo que sí que tenemos que utilizar siempre que usemos listas o diccionarios es el espacio de nombres: `System.Collections.Generic`. Así que modificaremos esas líneas para tener:

```
using UnityEngine;
using System.Collections.Generic;
```

Métodos y propiedades de una lista List<T>

Propiedades:

`int Count`
Contiene el número de elementos de la lista (equivalente al Length de los arrays).

Métodos:

`void Add(elemento)`
Añade `elemento` al final de la lista (equivalente al Push() de los arrays).

`void AddRange(coleccion)`
Permite añadir los elementos de un array o cualquier otra colección de datos que implemente la interfaz `IEnumerable`.

`void Clear()`
Elimina todos los elementos de la lista (`Count` pasa a ser 0).

`bool Contains(elemento)`
Devuelve `true` si la lista contiene el elemento especificado, `false` si no.

`int IndexOf(elemento)`
Devuelve el índice de la primera aparición de `elemento` en la lista.
Si `elemento` no está en la lista devuelve `-1`.

`int LastIndexOf(elemento)`
Devuelve el índice de la última aparición de `elemento` en la lista.
Si elemento no está en la lista devuelve `-1`.

`void Insert(posicion, elemento)`
Inserta `elemento` en la lista en la `posicion` especificada. Es un método lento. Si tenemos que hacer esta operación frecuentemente deberíamos utilizar otro tipo de colección (`LinkedList` o `Queue`)

`bool Remove(elemento)`
Elimina el primer `elemento` que encuentra en la lista. Si lo encuentra además de eliminarlo devuelve `true`. Si elemento no está en la lista devuelve false. El que devuelva un mooleano permite que hagamos que nuestro código reaccione al éxito o el fracaso de la operación.

`int RemoveAll(elemento)`
Elimina todas las apariciones de `elemento` la lista. Devuelve el número de elementos eliminados.

`void RemoveAt(int indice)`
Elimina el elemento que tiene el `indice` especificado.

`void RemoveRange(int indiceInicial, int noElementos)`
Elimina `noElementos` a partir de la posición especificada como `indiceInicial`.

`void Reverse()`
Invierte el orden de los elementos de la lista desde la que se invoca. Ejemplo: `enemies.Reverse();`

Además, tenemos el acceso por índice y la asignación por índice como en los arrays y con la misma sintaxis:

```
enemies[2].transform.position = ...;
...
enemies[0] = Instantiate(crazyEnemyPrefab);
```

Hay muchos más métodos de `List` que puedes encontrar descritos en ésta página: List<T> Class. Sin embargo, la descripción que hacen allí es mucho más técnica y la mayoría de los métodos que no he descrito aquí requieren una compresión mucho más avanzada del lenguaje.