



«Talento Tech»

Desarrollo de Videojuegos

# Unity 2D

Clase 08



«Talento Tech»

# **Clase N° 08 | Estructuras de datos**

## **Temario:**

- Lists
- Arrays
- Dictionary
- For
- ForEach

# Estructuras de datos

Las estructuras de datos son maneras organizadas de almacenar y manejar datos para que puedan ser utilizados de forma eficiente. Estas permiten a los programadores gestionar y manipular grandes volúmenes de información de forma efectiva, optimizando el rendimiento de las operaciones que se realizan sobre los datos.

Tienen varios tipos como: Lineales, No-lineales y Hash. En el curso estaremos incursionando en 2 Lineales; **Arrays** y **Lists**.

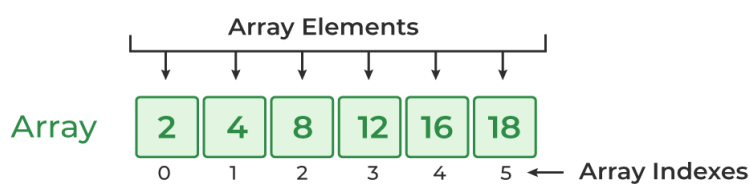
Se les dice lineales porque organizan los elementos de forma secuencial, es decir, cada elemento tiene un único predecesor y un único sucesor, excepto el primero y el último. Esta disposición en línea permite acceder a los elementos de manera ordenada.

## Array

### ¿Qué es un Array?

Permite almacenar una colección de **elementos** del mismo tipo en una secuencia continua en memoria. Cada **elemento** del array se puede acceder mediante un **índice**, lo que proporciona un acceso rápido y eficiente a los datos. El tamaño del Array es **Estático**.

Esto significa que nuestro array es un objeto que puede contener una determinada cantidad del mismo dato.



## Cómo se define y ejemplo de uso básico

```
public float[] a1;//Defino el Array de tipo Float

void Start()
{
    a1 = new float[5];//Asigno el tamaño del array
    //Asigno los valores por índice
    a1[0] = 32f;//en el índice 0, asigno el 32f
    a1[1] = 24f;
    a1[2] = 8f;
    a1[3] = 12f;
    a1[4] = 0f;
}
```

Las partes del Array se dividen en 3:

- **El tipo de dato:** en este caso float.
- **El nombre:** a1.
- **El tamaño:** lo que asignamos al escribir “**new float [5]**”.

Tengan en cuenta que el índice empieza siempre de 0, por lo tanto, si yo le doy un tamaño de 5, el índice llegará a 4.

Como verán, **primero** definimos la variable y en la misma podemos **asignarle el espacio** o podemos hacerlo dentro del Start(), como muestra el ejemplo.

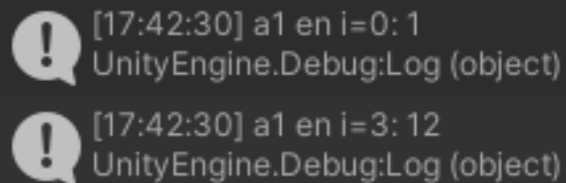
Debajo tenemos la asignación de datos, donde guardaremos cada dato deseado en el índice elegido.

```
a1[0] = 32f;//en el índice 0, asigno el 32f
a1[1] = 24f;
a1[2] = 8f;
a1[3] = 12f;
a1[4] = 0f;
```

Si queremos acceder a estos datos, debemos utilizar al array de la misma forma, con su índice:

```
Debug.Log("a1 en i=0: " + a1[0]);  
Debug.Log("a1 en i=3: " + a1[3]);
```

Viéndose de esta manera en consola:



```
[17:42:30] a1 en i=0: 1  
UnityEngine.Debug:Log (object)  
[17:42:30] a1 en i=3: 12  
UnityEngine.Debug:Log (object)
```

## Lists

### ¿Que es una list?

Una **List en C#** es una colección dinámica que permite almacenar una secuencia de elementos del mismo tipo. A diferencia de los arrays, las listas pueden crecer y decrecer en tamaño según sea necesario, lo que las hace más flexibles para la manipulación de datos.

Una característica importante es la cantidad de funciones/métodos que pueden usar con una lists, como Add, Remove, Sort, Contains, Clear, entre otros, permitiendo facilidades en la manipulación de datos.

### ¿Cómo se define? y ejemplo de uso básico

```
public class Clase8List : MonoBehaviour  
{  
    List<int> numeros = new List<int>() { 8, 23 };  
    List<string> nombres;  
  
    void Start()  
    {  
        nombres = new List<string>() { "Python", "Java" };  
    }  
}
```

En este código creamos 2 lists, una de tipo **int**, llamada “numeros” y otra de tipo **string**, llamada “nombres”.

En “números”, asignamos los valores al definirla.

En “nombres”, la asignamos en el Start().

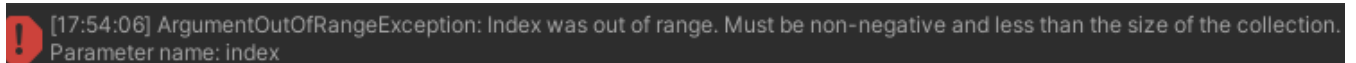
Las dos formas son posibles, *pero tengan en cuenta que el gasto de memoria se hace en dos momentos distintos. En uno el proceso se pide en la misma instancia de carga del proyecto/Script, en el otro, cuando el objeto aparece en escena y se llama al Start(). Tanto en variables como en esto, debemos tenerlo presente.*

Si queremos acceder a sus datos, lo hacemos igual que el Array. La llamamos y colocamos el valor del índice.

```
Debug.Log("valor en i=0: " + numeros[0]);  
Debug.Log("valor en i=1: " + numeros[1]);
```

Verán que si intentamos acceder al “índice 2”, nos tirara un error:

```
Debug.Log("valor en i=2: " + numeros[2]);
```



[17:54:06] ArgumentException: Index was out of range. Must be non-negative and less than the size of the collection.  
Parameter name: index

Si lo traducimos nos estaría diciendo que el “index” requerido está fuera de rango... Es decir, que NO existe.

## Añadiendo elementos (Add)

Si prestamos atención, en la lista no definimos su tamaño, sino que directamente le asignamos valores. Ahora **¿Cómo le podemos añadir más elementos?**

Esto será con la función “Add”, y es tan fácil como llamar a la función y pasarle el dato correspondiente.

```
numeros.Add(4);
```

Recordemos que SIEMPRE tiene que ser del mismo dato del que la definimos, en este caso un **int**.



## Remover Elemento (Remove)

Actualmente nuestra List se compone de 3 elementos:

numeros[0] = 8

numeros[1] = 23

numeros[2] = 4

¿Qué pasaría si deseamos eliminar alguno?

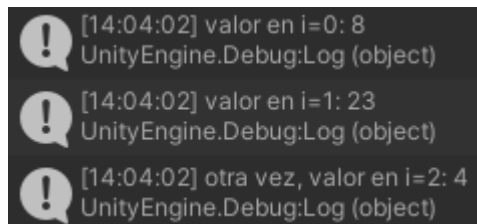
Por ahora usaremos 2 formas.

La primera es **Remove()** que removerá un elemento pasado como argumento. Por ejemplo:

```
numeros.Remove(23);

Debug.Log("valor en i=0: " + numeros[0]);
Debug.Log("valor en i=1: " + numeros[1]);
```

Antes de remover:

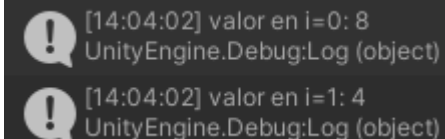


[14:04:02] valor en i=0: 8  
UnityEngine.Debug:Log (object)

[14:04:02] valor en i=1: 23  
UnityEngine.Debug:Log (object)

[14:04:02] otra vez, valor en i=2: 4  
UnityEngine.Debug:Log (object)

Después de remover:



[14:04:02] valor en i=0: 8  
UnityEngine.Debug:Log (object)

[14:04:02] valor en i=1: 4  
UnityEngine.Debug:Log (object)

Como podrán ver, el valor en el numeros[1], pasó a ser de 23 a 4, esto es porque removimos el 23 y el valor de numeros[2] (que era 4), se pasó al índice anterior.

La segunda forma que veremos es el **RemoveAt()**, que a diferencia de la anterior, remueve el valor del índice que coloquemos como argumento.

Por ejemplo, si ahora tenemos

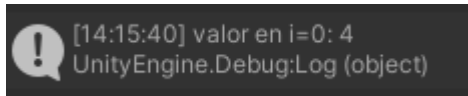
números[0] = 8

números[1] = 4

y si yo quiero remover el valor del **índice 0**, haria lo siguiente:

```
numeros.RemoveAt(0);
Debug.Log("valor en i=0: " + numeros[0]);
```

Dejando en consola este mensaje:



Al remover el valor del índice 0 (8), el 4 pasa a ser el único dato dentro de la lista.

## Ordenar Elementos (Sort)

Volviendo a nuestra lista de:

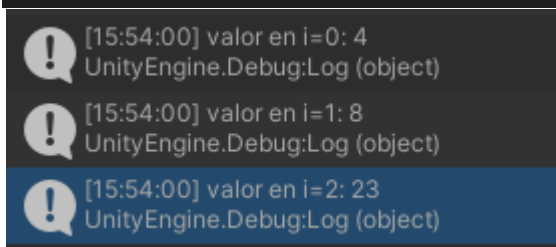
`numeros[0] = 8`

`numeros[1] = 23`

`numeros[2] = 4`

Suponiendo que deseamos ordenar los valores de nuestra lista, podemos utilizar el **Sort()**. Esta función se utiliza para ordenar los elementos de una lista en su lugar, es decir, modifica la lista original para que sus elementos queden en orden. Por defecto, ordena los elementos en orden ascendente, utilizando el método de comparación predeterminado de los elementos.

```
numeros.Sort();  
Debug.Log("valor en i=0: " + numeros[0]);  
Debug.Log("valor en i=1: " + numeros[1]);  
Debug.Log("valor en i=2: " + numeros[2]);
```



Podremos observar que fueron acomodados de mayor a menor.



## Vaciar Lista (Clear)

Si por algún motivo necesitamos eliminar cada elemento de nuestra lista, podemos hacerlo fácilmente con **Clear()**.

```
numeros.Clear();
```

## Cantidad de Elementos (Count)

**Count()**, es una propiedad utilizada para conocer el número total de elementos dentro de nuestra list. Esto nos será de mucha utilidad para el manejo general de los elementos.

```
int n = numeros.Count;  
Debug.Log("Cantidad de elementos: " + n);
```

Como es una **propiedad** y NO una función, tengan en cuenta que **no es invocable**. En este caso guardamos el valor resultante en una variable local.

## Dictionary (Diccionario)

En C#, un Dictionary es una colección que almacena pares de clave-valor. Cada valor en el diccionario se asocia con una clave única, lo que permite acceder rápidamente a los valores utilizando esas claves.

### Características principales:

- Claves únicas: No puedes tener dos entradas con la misma clave en un diccionario.
- Acceso rápido: Permite un acceso muy eficiente a los elementos mediante sus claves.
- Genéricos: Podés especificar el tipo de clave y el tipo de valor que contendrá el diccionario.

## Implementación

Los diccionarios tienen algo llamado **Key (Clave)** y **Value (Valor)**, siendo la forma en las que se estructuran sus datos, estos “clave-valor” mencionados anteriormente.

### Key (Clave):

- Es un identificador único para cada valor en el diccionario.
- Permite acceder, buscar o modificar el valor asociado.
- Las claves deben ser únicas dentro del mismo diccionario; no puedes tener dos entradas con la misma clave.
- El tipo de la clave se especifica al crear el diccionario.

### Value (Valor):

- Es el dato que se asocia a una clave específica.
- Puede ser de cualquier tipo, y múltiples claves pueden asociarse al mismo valor.
- Al igual que las claves, el tipo de valor se especifica al crear el diccionario.

Veamos un Ejemplo de su uso:

```
Dictionary<string, int> edades = new Dictionary<string, int>();

// Agregar elementos
edades.Add("Juan", 30); //La Key es "Juan" y el Value es 30
edades.Add("Camila", 25);
edades.Add("Felipe", 35);

// Acceder a un valor
Debug.Log("La edad de Juan es: " + edades["Juan"]);

// Comprobar si una clave existe
if (edades.ContainsKey("Camila"))
{
    Debug.Log("La edad de Camila es: " + edades["Camila"]);
}
```

```
}
```

## Bucle For

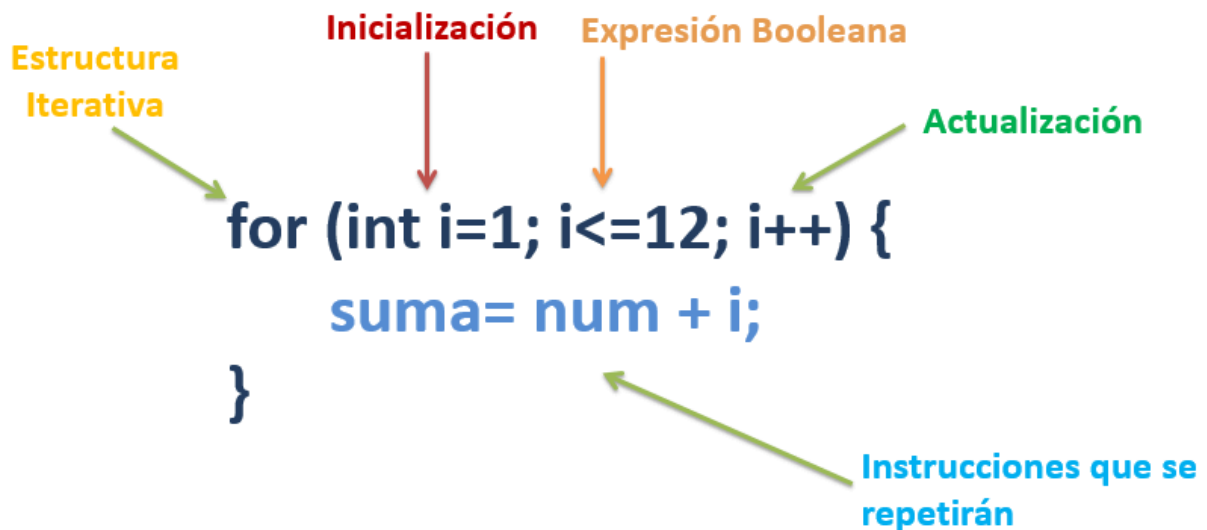
Para poder garantizar un mejor manejo de estas estructuras de datos, utilizaremos un **Bucle** llamado **For**. Pero ¿qué era un bucle?

Un bucle en programación es una estructura de control que permite **repetir** un bloque de código múltiples veces, **dependiendo de una condición**. Los bucles son fundamentales para automatizar tareas repetitivas y procesar colecciones de datos.

Dentro de ellos nos podemos encontrar con el While, DoWhile, For, For-each, entre otros.

### Implementación del For

El For es un bucle que se divide generalmente en 4 partes:



- El nombre de la estructura (for).
- La inicialización (`int i = 1`): El for es un bucle que trabaja en base a un variable “contador”, está mayormente es definida y asignada dentro de la misma herramienta.

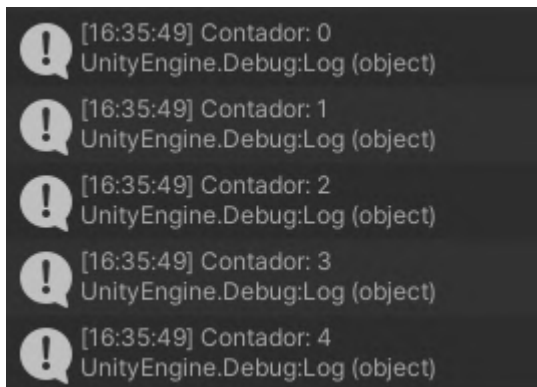
En base a esto pondremos desde donde queremos que arranque nuestro bucle.

- La “Expresión Booleana” o Condición ( $i \leq 12$ ): Esta parte determina el requerimiento necesario para que el bucle se siga ejecutando. En este caso, la condición es “mientras  $i$  sea menor o igual a 12”.
- Y por último la “Actualización o paso ( $i++$ ): Es lo que indicará de a cuanto suma nuestro contador cada vez que realiza una vuelta. En este caso sumaria de a 1.

## Ejemplo de uso básico

Crearemos un contador sencillo para mostrar el proceso que realiza este Bucle.

```
for (int i=0 ; i<5 ; i++)  
{  
    Debug.Log("Contador: " + i);  
}
```



Verán que la variable “ $i$ ”, que creamos en el mismo for, va sumando su valor hasta que la condición deja de cumplirse.

Si bien dice “Contador: 4”, observemos que empieza de 0, lo que significa que la parte de “ $i++$ ”, es decir, la actualización, la realiza cuando termina de ejecutar el código que tiene dentro, para luego preguntar si aun se sigue cumpliendo la condición.

Por lo tanto, si muestra el valor de  $i = 4$ , al terminar lo sumara, y cuando desea volver a realizar el ciclo, verá que  $i$  es igual a 5, terminando así el bucle y nunca mostrando su valor.

# Bucle ForEach.

Para algunos casos, como los diccionarios, vamos a necesitar específicamente un bucle llamado “ForEach”

```
foreach (tipo elemento in colección)
{
    // Acciones a realizar con cada elemento
}
```

Lo que hace es repasar la estructura colocada, utilizando una variable que irá adoptando el valor de cada elemento, sea del tipo que sea.

Así será que irá revisando 1 por 1 cada elemento dentro de nuestra “colección” y podremos acceder a ellos fácilmente.

Veamos un ejemplo con el diccionario creado anteriormente:

```
Dictionary<string, int> edades = new Dictionary<string, int>();

// Agregar elementos
edades.Add("Juan", 30); //La Key es "Juan" y el Value es 30
edades.Add("Camila", 25);
edades.Add("Felipe", 35);
// Iterar sobre el diccionario
foreach (var valores in edades)
{
    Debug.Log(par.Key + "tiene" + par.Value + "años");
}
```

## Ejemplo de uso Práctico (Inventario Simple)

Los verdaderos inventarios en los juegos **NO** son algo sencillo de reproducir, mucho menos si aun no tenemos el conocimiento necesario para gestionarlo.

Por eso, en este caso vamos a hacer una versión reducida, simplificada, pero que sigue manejando algunas lógicas del mismo y a la vez será adaptable para incorporar nuevos ítems.

## Script ItemManager

Para empezar, crearemos un Script llamado ItemManager

```
public class ItemManager : MonoBehaviour{
```

*El concepto de “Manager” es algo muy usado en programación, son aquellos script que “manejan” el funcionamiento de ciertos aspectos del juego. Cande destacar que por ahora esto, de “Manager” no tiene más que el nombre y veremos una pequeña introducción a ellos en clases futuras.*

Continuando con su código:

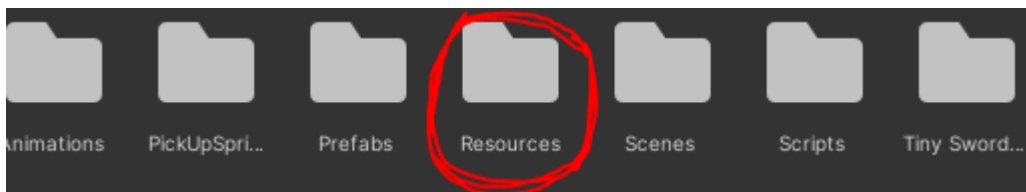
Creamos la lista que contendrá mis items y mi diccionario que se encargará de almacenarlos y organizar sus cantidades.

```
public List<GameObject> gameItems;  
// Creamos el Diccionario que tendrá nuestro inventario  
public Dictionary<GameObject, int> inventory = new Dictionary<GameObject, int>();
```

Seguiremos con el Start(), asignando a nuestra Lista “gameItems” todos los items “agarrable” que queramos.

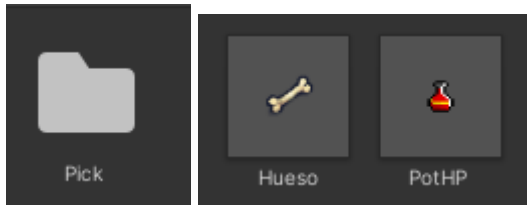
Para lograr que funcione debemos pasarle **DE DONDE** queremos que cargue nuestros objetos. ESto lo podemos lograr haciendo una lista pública y *pasar prefab por prefab* o le diremos de donde puede cargar cada objeto simplemente haciendo lo siguiente:

**Primero:** crearemos una carpeta que el sistema reconocerá por Default en base a su nombre. Será llamada **“Resources”** (recursos).



**Segundo:** podremos organizar cualquier recurso que vayamos a necesitar que Unity ubique por su cuenta.

Así, dentro de la misma, crearemos otra más con el nombre que queramos. Yo le puse “Pick” y dentro guarde los prefabs que quiero que sean PickUps:



**Tercero:** pondremos en el código:

```
void Start(){  
    gameItems = new List<GameObject>(Resources.LoadAll<GameObject>("Pick"));
```

Si lo analizan, verán que a nuestra variable, le asignamos una nueva lista y esta sacará su valor de la carpeta Resources, cargando todos los gameObject (“LoadAll”), dentro de la carpeta “Pick”.

Seguiremos armando un **For** que iteara sobre nuestra **lista** e irá añadiendo a nuestro Diccionario, cada **GameObject** como una **Key**, junto con con un **value** de “0”, que lo usaremos para representar la cantidad “almacenada” de ese objeto.

```
for(int i = 0 ; i<gameItems.Count; i++ )  
{  
    inventory.Add(gameItems[i], 0);  
}
```

**Cuarto:** y por último, usando un ForEach mostraremos en consola los datos guardados

```
foreach (KeyValuePair<GameObject, int> items in inventory) {  
    print("Tenes " + items.Value + " " + items.Key);  
}
```



Código Completo:

```
public List<GameObject> gameItems;
// Creamos el Diccionario que tendrá nuestro inventario
public Dictionary<GameObject, int> inventory = new Dictionary<GameObject, int>();
void Start(){
    gameItems = new List<GameObject>(Resources.LoadAll<GameObject>("Pick"));

    for(int i = 0 ; i<gameItems.Count; i++ )
    {
        inventory.Add(gameItems[i], 0);
    }
    //gameItems.Clear();

    foreach (KeyValuePair<GameObject, int> items in inventory) {
        print("Tenes" + items.Value + " " + items.Key);
    }
}
```

## Script Item (El Pickup o Agarrable)

Ahora crearemos otro Script llamado "Item" que tendrá un código necesario para cada "PickUp" o item agarrable.

Pondremos el código completo y luego analizaremos parte por parte:

```
public class Item : MonoBehaviour {
    public int myCount = 1;
    [SerializeField]
    private string objID;
    private void OnTriggerEnter2D(Collider2D collision) {
        if (collision.CompareTag("Player")) {
            ItemManager inv = collision.GetComponent<ItemManager>();
            foreach (KeyValuePair<GameObject, int> items in inv.inventory) {
                Item invID = items.Key.GetComponent<Item>();
                if (invID.objID == objID) {
                    Debug.Log("Obtuve un: " + gameObject.tag);
                    inv.inventory[items.Key] += myCount;
                }
            }
        }
    }
}
```

```

        print("You have " + inv.inventory[items.Key] + " " + items.Key);
        Destroy(gameObject, 0.2f);
        break;
    }
}
}
}

```

Variable “myCount” que dirá cuántos de mis items contiene este objeto y la variable “objID” que se hará cargo de **identificar** que tipo de ítem manejamos.

```

public int myCount = 1
[SerializeField]
private string objID;;

```

Dentro de un OnTriggerEnter2D, preguntamos si está chocando contra el “Player”

```

private void OnTriggerEnter2D(Collider2D collision) {
    if (collision.CompareTag("Player")) {

```

Creamos una variable de **referencia a nuestro ItemManager**, que, en este caso, “casualmente” se encuentra dentro de nuestro personaje

```

ItemManager inv = collision.GetComponent<ItemManager>();

```

Ponemos un ForeEach para iterar nuestro diccionario:

```

foreach (KeyValuePair<GameObject, int> items in inv.inventory) {

```

Tengan en cuenta que en C#, un **KeyValuePair<TKey, TValue>** es una estructura que representa un par de **clave(Key) y valor(Value)**, donde TKey es el tipo de la clave y TValue es el tipo del valor. Se utiliza principalmente para trabajar con colecciones que almacenan datos en pares, como los diccionarios.

En este caso, cada elemento se irá guardando en la variable **items**, haciendo que esta tenga como **Key** el GameObjects/item que nos interesa y cómo **Value**, la cantidad que tenemos guardada en nuestro “inventario”

Dentro de este, haremos una variable que guarde como referencia el componente "Item" de nuestro Diccionario y colocaremos un if para verificar si el elemento actual, guardado en la variable "items" gracias a nuestro **ForEach**, tiene como **Key** (que pusimos que sean GameObjects), el mismo **objID** que este GameObject

```
Item invID = items.Key.GetComponent<Item>();  
    if (invID.objID == objID) {
```

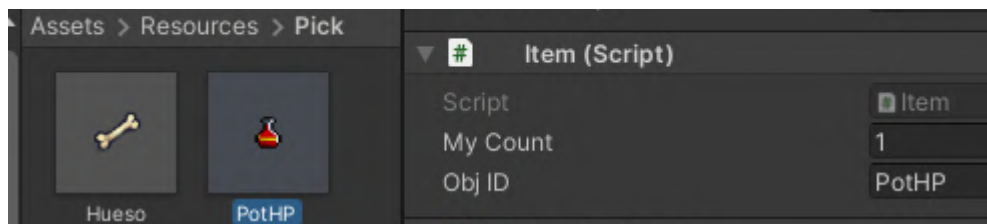
Accedemos al diccionario para sumarle la cantidad de ítems agarrados y le diremos a nuestro objeto que se destruya para simular un "Agarre este ítem y por lo tanto ya no está más en el suelo"

```
        Debug.Log("Obtuve un: " + objID);  
        inv.inventory[items.Key] += myCount;  
        Debug.Log("tenes " + inv.inventory[items.Key] + " " + items.Key);  
        Destroy(gameObject, 0.2f);  
        break;
```

Y terminamos usando el "Break;".

¿**Qué hace el Break?** Le indica a esta iteración/bucle que debe de terminar, sino seguiría revisando el diccionario habiendo modificado parte del mismo y tirándonos un error.  
(Los Debug.Log usados son para que nos muestre en consola el ítem que agarramos y cuántos de ellos tenemos actualmente)

**Por último** sobre este Script, **asegúrense** de ir a cada Prefab y asignarle un ID para reconocerlos:



## Script UseItems (Usar los ítems)

Llegando a la recta final de este ejemplo, nos queda el uso de estos ítems. En este caso haremos funciones distintas según el uso que deseemos

```
public class UseItem : MonoBehaviour {
```

```

[SerializeField]
private ItemManager im;
[SerializeField]
private int vida = 50;

void Start() {
    im = GetComponent<ItemManager>();
}

void Update() {
    UsingHealingPotion2();
}

private void UsingHealingPotion2() {
    if (Input.GetKeyDown(KeyCode.Alpha2)) {
        foreach (KeyValuePair<GameObject, int> items in im.inventory) {
            Item obj = items.Key.GetComponent<Item>();
            if (obj.objID == "PotHP") {
                if (items.Value > 0) {
                    vida += 10;
                    Debug.Log("Vida actual: " + vida);
                    im.inventory[items.Key] -= 1;
                    break;
                }
            }
        }
    }
}
}
}
}
}

```

Lo veremos por partes:

**En primer lugar** crearemos las variable:

```

[SerializeField]
private ItemManager im;
[SerializeField]
private int vida = 50;

```

La variable "im" será nuestra referencia al ItemManager y la variable vida... nuestra vida.

**Después** asignaremos nuestra variable im y en el Update() llamaremos a la función que vamos a crear.

```

void Start() {
    im = GetComponent<ItemManager>();
}

```

```

}
void Update() {
    UsingHealingPotion2();
}

```

**Seguimos** creando la función que luego será llamada en el Update(), y le pondremos de nombre algo asociado al ítem que queremos usar:

```

private void UsingHealingPotion2()
{

```

Pondremos un if para que detecte cuando apretamos una tecla, en este caso el **número 2**

```

if (Input.GetKeyDown(KeyCode.Alpha2))
{

```

Avanzamos con el mismo `ForEach` del código anterior:

```

foreach (KeyValuePair<GameObject, int> items in im.inventory)
{

```

Crearemos y asignaremos nuestra **variable** de referencia, que guardará el componente **Ítem** puesto en el **GameObject** de la **Key** de esta iteración. Así podremos obtener su **objID**

```

Item obj = items.Key.GetComponent<Item>();

```

Preguntaremos si tenemos un **objID "PotHP"**, que es el colocado en **nuestra poción** de vida. Y si lo encuentra, le preguntaremos el **Value** en esa **Key**, para saber si poseemos "Stock"/Existencias de ese Ítem ahora mismo.

```

if (obj.objID == "PotHP") {
    if (items.Value > 0) {

```

Si tenemos este ítem, nos aumentaremos la vida. Acá lo hacemos llamando a la función **"Heal()"**(*Curar*) y pasándole cuanto debe curarse. También se puede hacer sumando directamente el valor.

```

Heal(10); // o vida = 10;
Debug.Log("Vida actual: " + vida);

```

Cómo usamos el ítem, le restamos en 1 su cantidad (el **Value** de la **Key**)

```

im.inventory[items.Key] -= 1;

```

```
break;
```

Y terminaremos por usar el “break;”

De esta manera, un poco extensa, pero bastante sencilla, usando lo aprendido podremos tener un mini inventario que irá guardando cada ítem que Guardemos en la carpeta “**Resources/Pick**” y asignando su comportamiento de **Ítem**

Tengan en cuenta que ahora mismo, el objeto no tiene un “comportamiento de uso” específico, sino que depende de nuestro Script “UseItem”. Esto NO es lo ideal. Lo mejor SIEMPRE es que cada objeto se ocupe de su comportamiento. Imaginemos que tengo 10mil objetos. Tendría que crear un Código de más de 10mil líneas para abarcar el comportamiento de cada objeto.

***Esto lo resolveremos en clases siguientes.***

## Ejercicios prácticos:

### ¡PickMe!

Utilizando el Ejemplo Práctico, crear otro ítem Agarrable/PickUp. Recuerden que para eso deben guardar el Prefab en la carpeta “**Pick**”, asignarle el componente “**Item**”, poner un “**objID**” y crear un comportamiento de uso en el Script “**UseItem**”



**Buenos Aires**  
*aprende*  
Agencia de Habilidades para el Futuro

**BA** Buenos  
Aires  
Ciudad