Programación 3

Orientación a objetos con **C#**

4. Excepciones, Genéricos, Colecciones

maurogullino@gmail.com

Named arguments

```
static void Main()
{
      //prueba("Pepe", 20, 300.40);
      prueba(edad: 20, nombre: "Pepe", sueldo: 300.40);
}

static void prueba(string nombre,
                                 int edad, double sueldo)
{
      Console.WriteLine(nombre);
      Console.WriteLine(edad);
      Console.WriteLine(sueldo);
}
```

ref arguments (pasaje por referencia)

```
static void prueba(ref int i, ref string s1, ref string s2)
{
     i = 44;
     s1 = "retorname";
}

static void Main()
{
     int valor = 10;
     string str1 = "hola", str2="chau";
     prueba(ref valor, ref str1, ref str2);
}
```

out arguments (variables de salida)

```
static void prueba(out int i, out string s1, out string s2)
{
     i = 44;
     s1 = "retorname";
     s2 = null;   //qué pasa si lo comentamos?
}

static void Main()
{
     int valor;
     string str1, str2;
     prueba(out valor, out str1, out str2);
}
```

ref

· provoca un pasaje por referencia
· debe inicializarse antes de la llamada
· la fc invocada puede no tocarla

out

· provoca un pasaje por referencia
· la iniciación previa no es obligatoria
· la fc invocada está obligada a asignarla
· es un “ref especializado”: seguridad y legibilidad

out: ejemplo de aplicación

```
string origen = "666";
int num;

if (Int32.TryParse(origen, out num))
     Console.WriteLine($@"Convertido
                          '{origen}' to {num}");
else
     Console.WriteLine($@"No se pudo convertir
                               '{origen}'");
```

Otro ejemplo (con objetos)

```
static void prueba(Premio c) {
    Premio d = new Premio();
    d.Nombre = "Perro salchicha";
    c = d;
}

static void Main() {
    Premio a = new Premio();
    Premio b = a;
    a.Nombre = "LED 40 pulgadas";

    prueba(a);
    Console.WriteLine(a.Nombre);
    Console.WriteLine(b.Nombre);
    Console.WriteLine(a==b);
}
```

```
static void prueba(ref Premio c)  {
    Premio d = new Premio();
    d.Nombre = "Perro salchicha";
    c = d;
}

static void Main()  {
    Premio a = new Premio();
    Premio b = a;
    a.Nombre = "LED 40 pulgadas";

    prueba(ref a);
    Console.WriteLine(a.Nombre);
    Console.WriteLine(b.Nombre);
    Console.WriteLine(a==b);       //!!!
}
```

Excepciones

```
class Deuda {
    public double ObtenerDeudaMensual(ushort mes)
    {
        if (mes < 1 || mes > 12)
        {
            //qué hacemos???
        }

        return 3333;
    }
}
```

```
class Deuda {
    public double ObtenerDeudaMensual(ushort mes)
    {
        if (mes < 1 || mes > 12)
        {
            throw new MesInexistente();
        }

        return 3333;
    }
}

class MesInexistente : Exception { }
```

Exceptions

- representan una situación de error (es un objeto)
- ponen al programa en un estado “de excepción”
- el runtime buscará alguien que se “haga cargo” del problema
- si nadie se hace cargo se finaliza el programa (!)

```
static void Main()  {
    Deuda d = new Deuda();

    try {
       Console.WriteLine(d.ObtenerDeudaMensual(13));
    }
    catch(MesInexistente obj1) {
       Console.WriteLine("El mes indicado no existe");
    }
    catch(Exception obj2) {
       Console.WriteLine("Ocurrió un error desconocido");
    }

}
```

Genéricos

```
static void Main() {
    int x = 5, y = 10;
    IntercambiadorDeInts(ref x, ref y);
}

static void IntercambiadorDeInts(ref int a, ref int b)
{
    int c = a;
    a = b;
    b = c;
}
```

```
static void Main() {
    int x = 5, y = 10;
    Intercambiador<int>(ref x, ref y);
}

static void Intercambiador<T>(ref T a, ref T b)
{
    T c = a;
    a = b;
    b = c;
}
```

```
class Pilita<ZZ>  {
     private ZZ[] datos;
     private int tope=0;

     public Pilita() {
         datos = new ZZ[10];
     }

     public void poner(ZZ x)  {
         datos[tope] = x;
         tope++;
     }

     public ZZ sacar()  {
         tope--;
         return datos[tope];
     }

 }
```

```
static void Main() {

      Pilita<int> pila = new Pilita<int>();

      pila.poner(4); pila.poner(5);

      Console.WriteLine(pila.sacar());
      Console.WriteLine(pila.sacar());

}
```

Colecciones

Colecciones

· más flexibles que los arrays (tamaño, acceso)
using System.Collections.Generic;

· más utilizadas: *List, Dictionary, Queue* (cola)

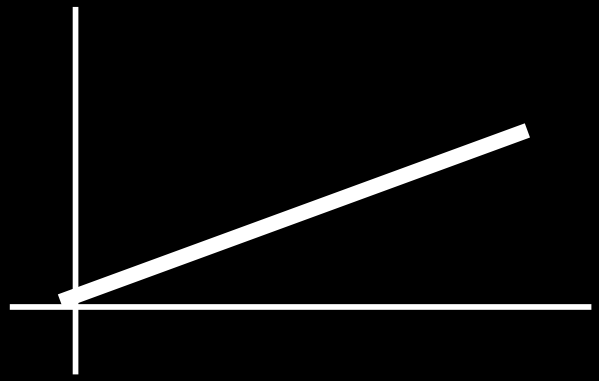
· pueden existir consideraciones especiales (concurrencia, rendimiento)

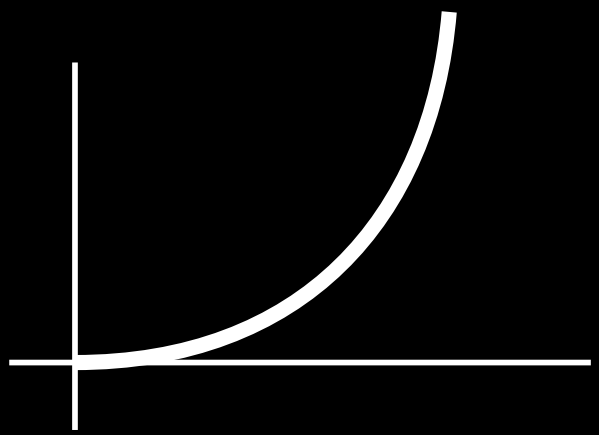
· ArrayList es bibliografía antigua (no Generic)

Big-O notation

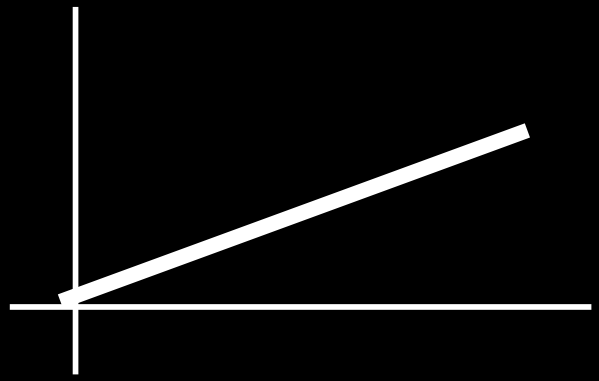
· describe la performance de un algoritmo
· cómo se comporta al incrementarse los datos
· generalmente se considera el **tiempo** de cálculo

O(1) tiempo constante *(ojo! capaz muy grande)*
O(n) crecimiento lineal
$O(n^2)$ crecimiento cuadrático
O(log n) crecimiento logarítmico

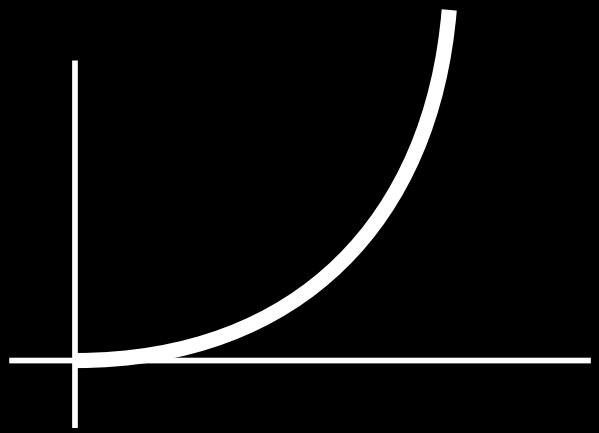




O(1)



O(n)



$O(n^2)$

O(log n)

List<T>

- es un array “dinámico”

```
var alumnos = new List<string>();
alumnos.Add("Pepe");
alumnos.Add("Juan");
alumnos.Add("Maria");

foreach (var al in alumnos)
{
    Console.WriteLine(al);
}
```

```
List<string> alumnos = new List<string>();
alumnos.Add("Pepe");
alumnos.Add("Juan");
alumnos.Add("Maria");
alumnos.Remove("Juan");
alumnos.Insert(1, "Segundo");

for (int i = 0; i < alumnos.Count; i++)
{
    Console.WriteLine(alumnos[i]);
}

Console.WriteLine(alumnos.Contains("Carlitos"));
Console.WriteLine(alumnos.IndexOf("Segundo"));
```

```
List<int> numeros = new List<int>();
for (int i = 0; i <= 20; i++)
{
    numeros.Add(i);
}

for (int i = numeros.Count - 1; i >= 0; i--)
{
    if (numeros[i] % 2 == 1)
    {
        numeros.RemoveAt(i);
    }
}
```

List<T>

- agregar o borrar al inicio/medio: O(n)
- agregar o borrar al final: O(1)
- acceso random: O(1)
- buscar un elemento: O(n)

Queue<T>

- cola, colección FIFO
- implementado con un array

```
Queue<string> numeros = new Queue<string>();
numeros.Enqueue("uno");
numeros.Enqueue("dos");
numeros.Enqueue("tres");

while (numeros.Count > 0)
      Console.WriteLine(numeros.Dequeue());
```

```
Queue<string> numeros = new Queue<string>();
numeros.Enqueue("uno");
numeros.Enqueue("dos");
numeros.Enqueue("tres");

// se pueden enumerar sin alterarlo
foreach (string n in numeros)
{
      Console.WriteLine(n);
}

Console.WriteLine(numeros.Contains("dos"));

//mirar el próximo, sin desencolarlo
Console.WriteLine(numeros.Peek());

numeros.Clear();
```

Queue<T>

· agregar en cola:
O(1) mejor caso
O(n) peor caso (mover todo el array interno)

· borrar siguiente: O(1)

· acceso random: no disponible

LinkedList<T>

- la querida lista (doblemente) enlazada

```
LinkedList<string> frase = new LinkedList<string>();
frase.AddFirst("mundo");
frase.AddFirst("hola");

foreach (string pal in frase)
{
     Console.Write(pal + " ");
}
```

```
LinkedList<string> frase = new LinkedList<string>();
frase.AddFirst("mundo");
frase.AddFirst("hola");
frase.AddLast("cruel");
frase.RemoveFirst();
frase.AddFirst("adios");

LinkedListNode<string> busq = frase.Find("mundo");
frase.AddAfter(busq, "tan");

Console.WriteLine(busq.Next);
Console.WriteLine(busq.Next.Value);
Console.WriteLine(frase.Count);
```

LinkedList<T>

- agregar en cualquier posición: O(1)
- borrar en cualquier posición: O(1)
- acceso random: O(n)
- buscar un elemento: O(n)

Dictionary<TK, TV>

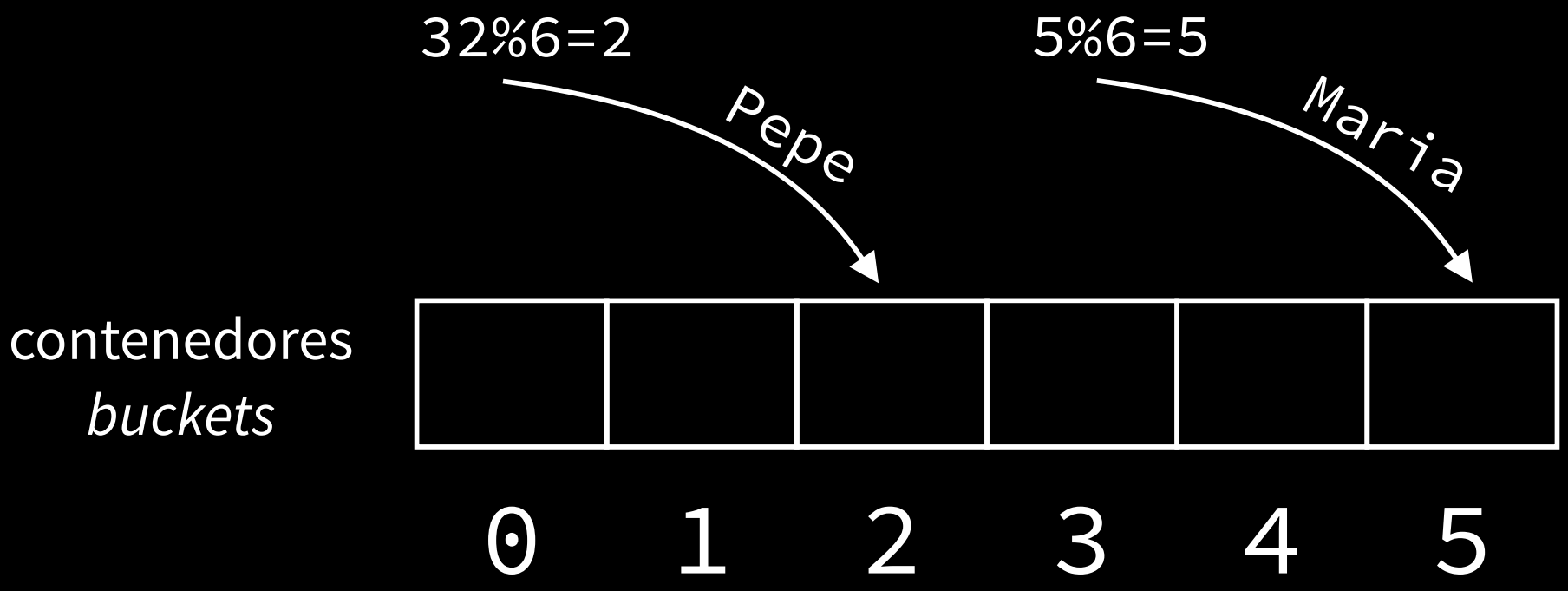
- colección de “pares clave/valor” *key/value*
- las claves no se pueden repetir

```
var clientes = new Dictionary<int, string>();
clientes.Add(32, "Pepe");
clientes.Add(2, "Juancito");
clientes.Add(4567, "Maria");

Console.WriteLine(clientes[2]); //ojo! parece vector
```

```
var clientes = new Dictionary<int, string>();
clientes.Add(32, "Pepe");
clientes.Add(2, "Juancito");
clientes.Add(4567, "Maria");
clientes.Add(111, "Leopoldo");
clientes.Remove(2);
clientes[32] = "Pepe II";

foreach(KeyValuePair<int,string> cv in clientes)  {
       Console.WriteLine(cv.Key + ": " + cv.Value);
}

Console.WriteLine(clientes[555]);  //ojo !
Console.WriteLine(clientes.ContainsKey(555));

if (clientes.TryGetValue(32, out string val))  {
       Console.WriteLine("Valor es: {0}", val);
}
else {
       Console.WriteLine("Clave no encontrada");
}
```

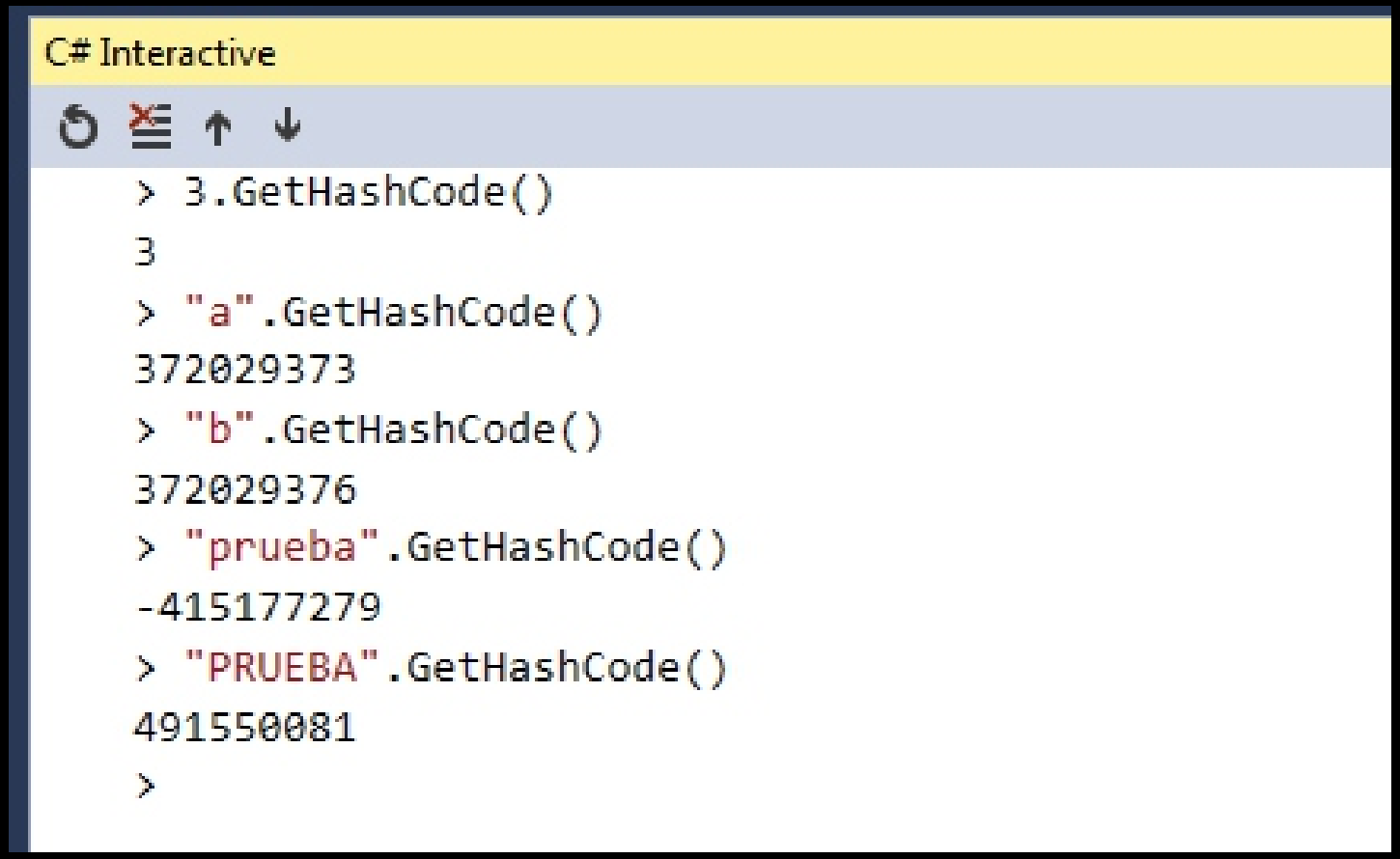
Diccionarios como tablas hash

```
var clientes = new Dictionary<int, string>();
clientes.Add(32, "Pepe");
clientes.Add(5, "Maria");
```



Tablas hash

- debe haber una *función hash*, que se aplica a las claves para obtener un número
- el hash determina a qué contenedor se envían los valores
- en cada contenedor hay una lista enlazada


C# Interactive
> 3.GetHashCode()
3
> "a".GetHashCode()
372029373
> "b".GetHashCode()
372029376
> "prueba".GetHashCode()
-415177279
> "PRUEBA".GetHashCode()
491550081
>

Dictionary<T, T>

- agregar y borrar: O(1) *depende de las colisiones
- acceso random: O(1) *depende de las colisiones
- buscar un elemento: O(n)

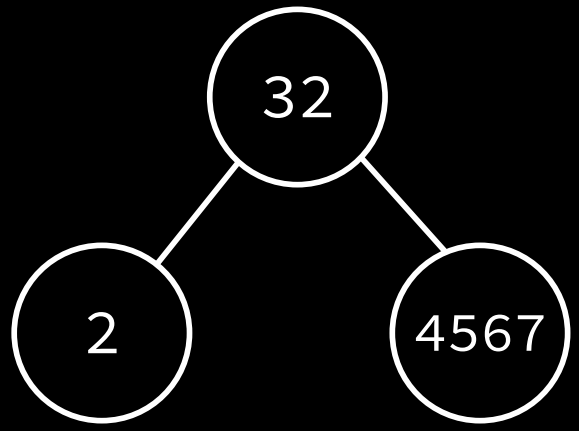
SortedDictionary<TK, TV>

- diccionario que se mantiene ordenado por clave
- en un Dictionary (*hash table*) el orden depende de la implementación y no está garantizado

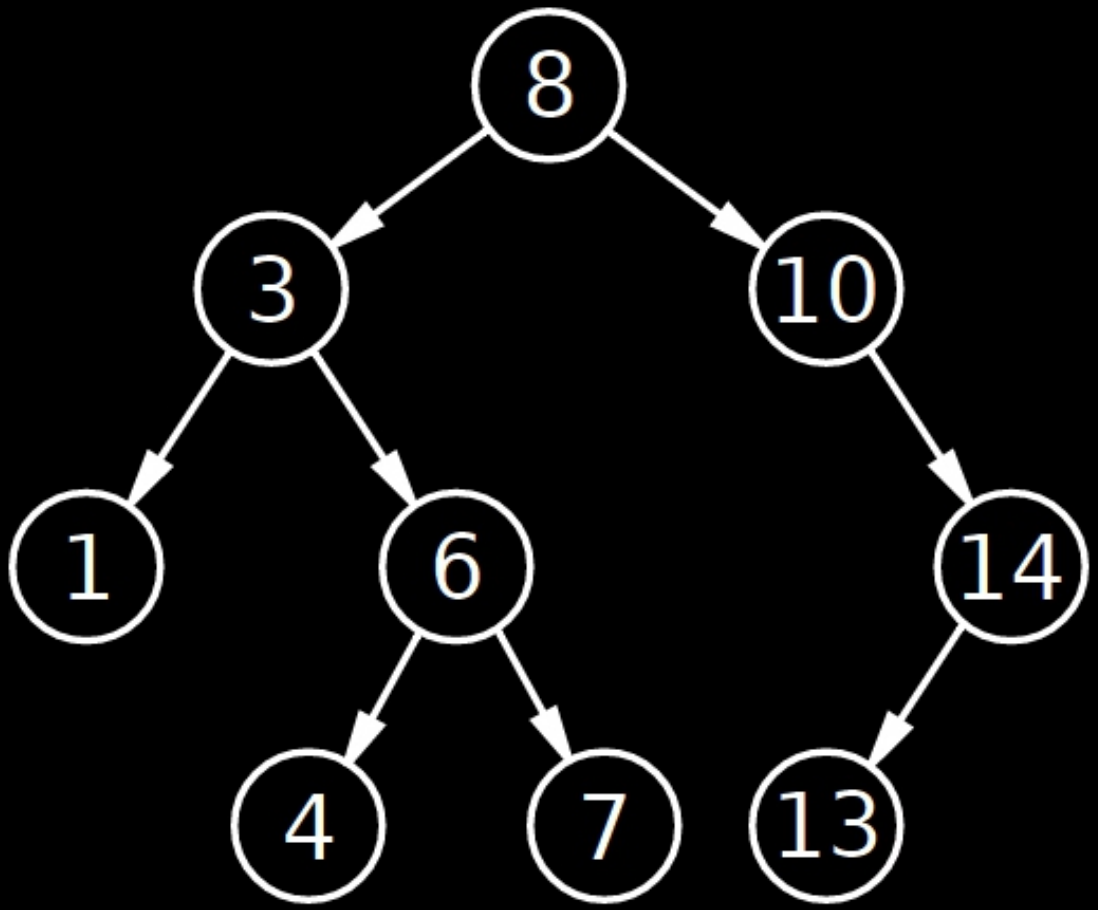
```
var clientes = new SortedDictionary<int, string>();
clientes.Add(32, "Pepe");
clientes.Add(2, "Juancito");
clientes.Add(4567, "Maria");

foreach(var c in clientes)  {
      Console.WriteLine(c.Key + " " + c.Value);
}
```

SortedDictionary como *Binary Search Tree* (BST)

```
var clientes = new SortedDictionary<int, string>();
clientes.Add(32, "Pepe");
clientes.Add(2, "Juancito");
clientes.Add(4567, "Maria");
```



Ejemplo de *Binary Search Tree* (BST)



Recorrido *in-orden*

- izquierda
- centro
- derecha

SortedDictionary<T, T>

- agregar y borrar: O(log n)
- acceso random: O(log n)
- buscar un elemento: O(log n)

Resumen

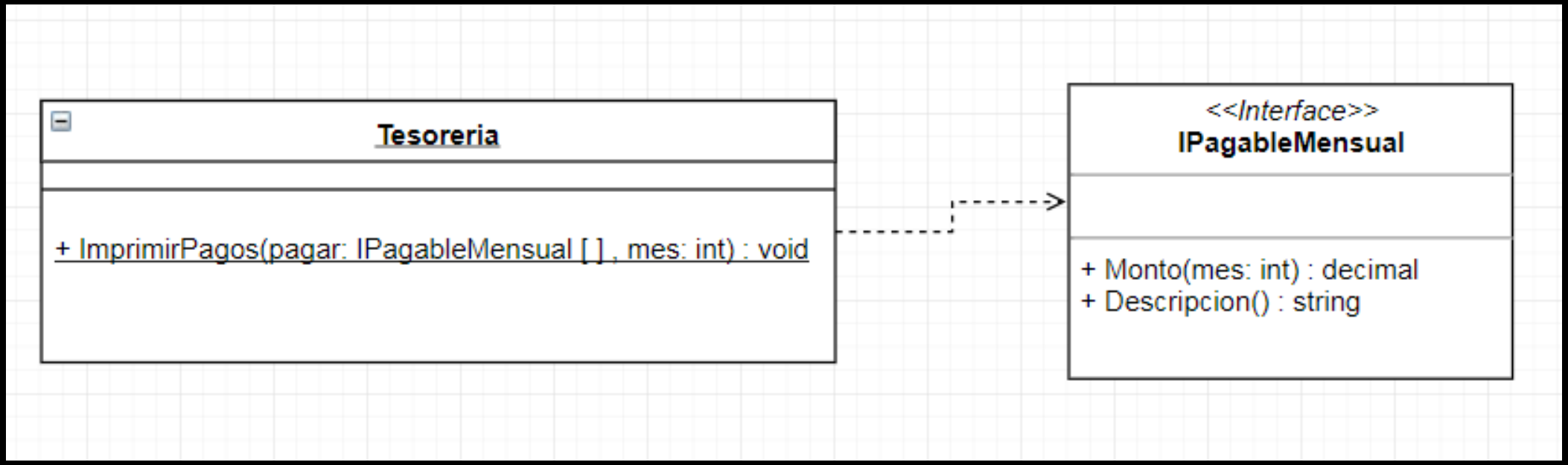
Colección .net	*Estructura de datos subyacente*
List	Vector
Queue	Vector
LinkedList	Lista enlazada
Dictionary	Hash table
SortedDictionary	Árbol binario

Comparación con C++ *Standard Template Library*

Colección .net	*C++ STL*
List<T>	std::vector<T>
Queue<T>	std::queue<T>
LinkedList<T>	std::list<T>
Dictionary<TK, TV>	std::unordered_map<TK, TD>
SortedDictionary<TK, TV>	std::map<TK, TD>

Revisita a UML

Diagrama de clases

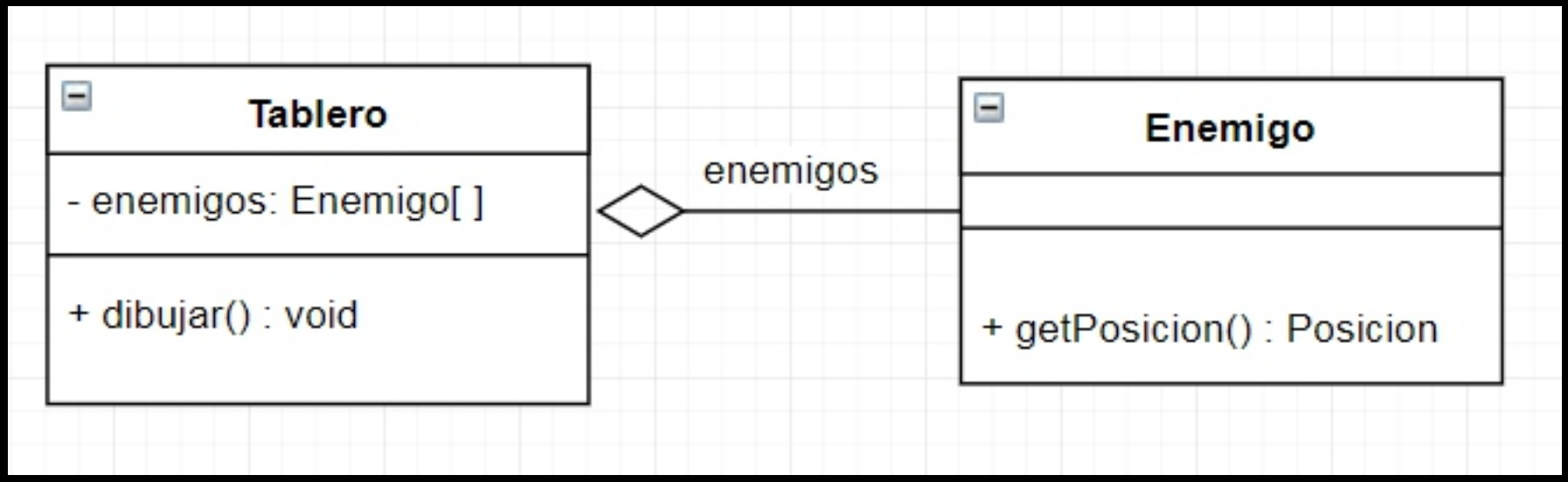


relación de dependencia
"usa un"
(interacción por corto tiempo)

Otras relaciones en UML

Aggregation

· relación de más largo plazo que *dependency*
· los objetos se pueden destruir independientemente
· se puede leer como “conoce unos...”

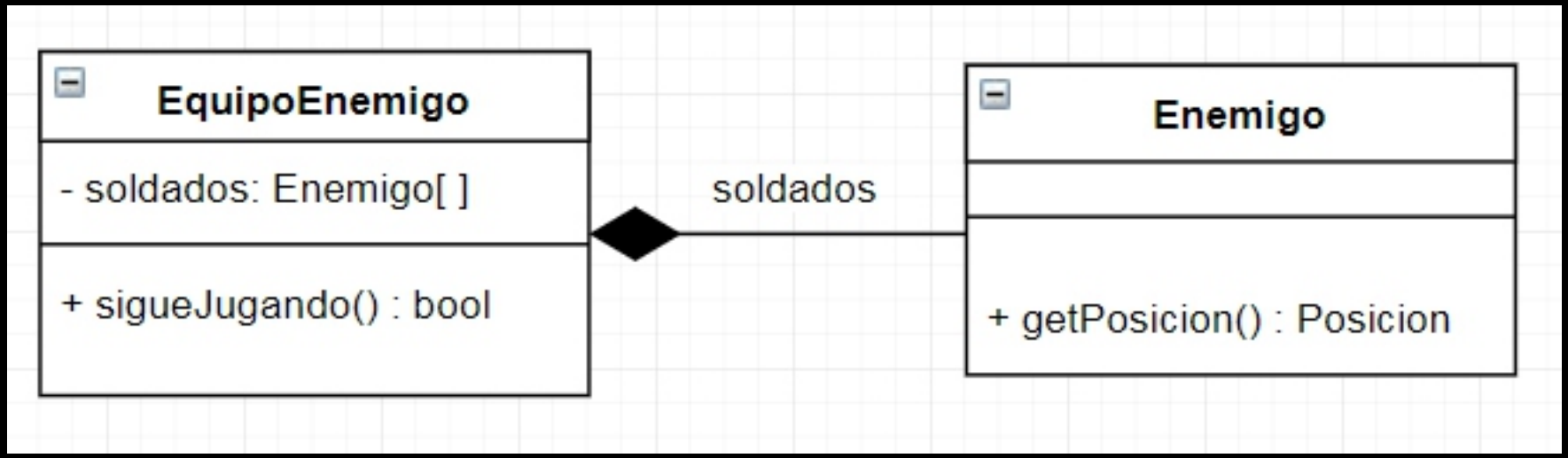


draw.io

Otras relaciones en UML

Composition

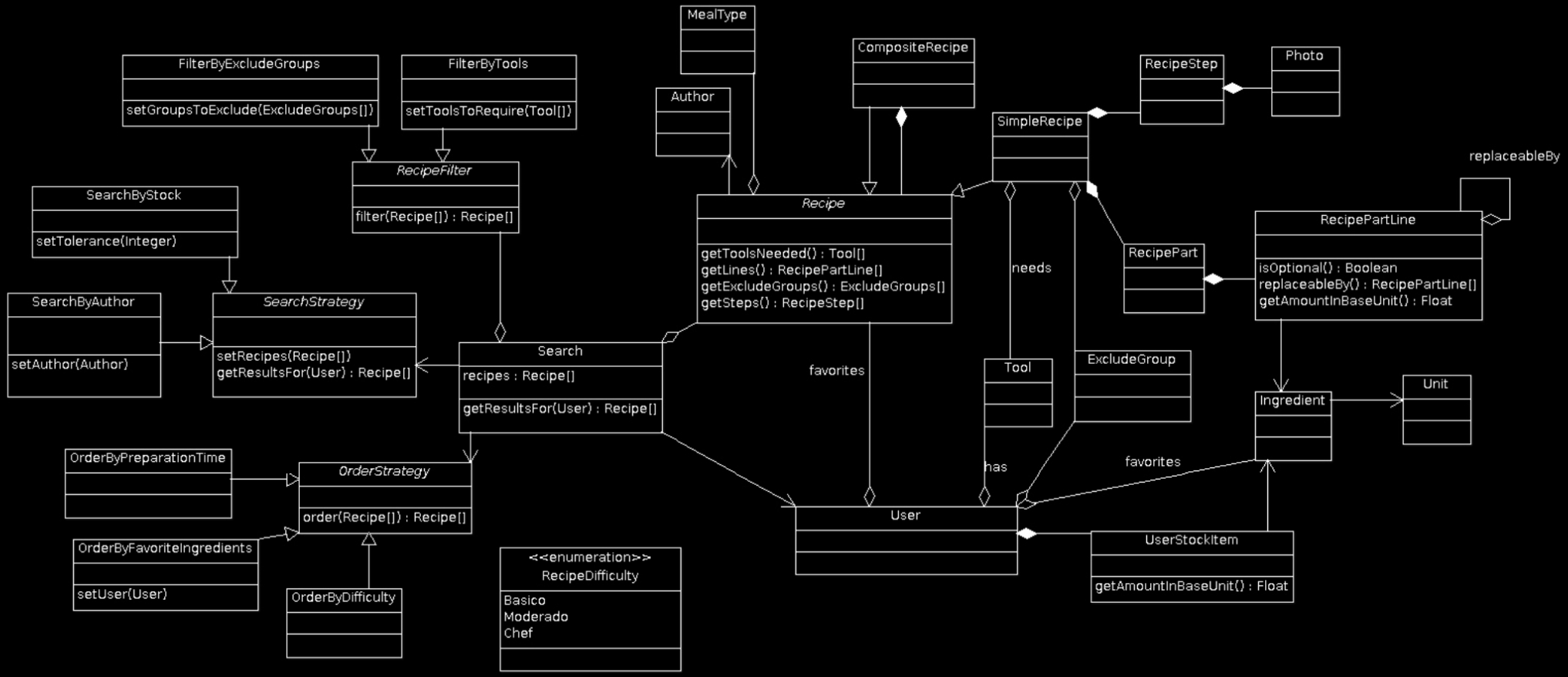
· la relación más fuerte
· los objetos son dependientes
· se puede leer como “está formado de...”



draw.io

Ejemplos en código

```
public class A {   //dependencia
    void Met(B b) {
    }
};

public class A {   //agregación
    private B b;
    Met(B z) {
       this.b = z;
    }
}

public class A {   //composición
    private B b = new B();
}
```


MealType
FilterByExcludeGroups
setGroupsToExclude(ExcludeGroups[])
FilterByTools
setToolsToRequire(Tool[])
CompositeRecipe
RecipeStep
Photo
Author
SimpleRecipe
replaceableBy
RecipeFilter
filter(Recipe[]) : Recipe[]
SearchByStock
setTolerance(Integer)
Recipe
getToolsNeeded() : Tool[]
getLines() : RecipePartLine[]
getExcludeGroups() : ExcludeGroups[]
getSteps() : RecipeStep[]
needs
RecipePart
RecipePartLine
isOptional() : Boolean
replaceableBy() : RecipePartLine[]
getAmountInBaseUnit() : Float
SearchByAuthor
setAuthor(Author)
SearchStrategy
setRecipes(Recipe[])
getResultsFor(User) : Recipe[]
Search
recipes : Recipe[]
getResultsFor(User) : Recipe[]
favorites
Tool
ExcludeGroup
Ingredient
Unit
OrderByPreparationTime
OrderStrategy
order(Recipe[]) : Recipe[]
has
favorites
User
OrderByFavoriteIngredients
setUser(User)
OrderByDifficulty
<<enumeration>>
RecipeDifficulty
Basico
Moderado
Chef
UserStockItem
getAmountInBaseUnit() : Float

Ejercicio

facturas e items