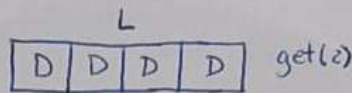
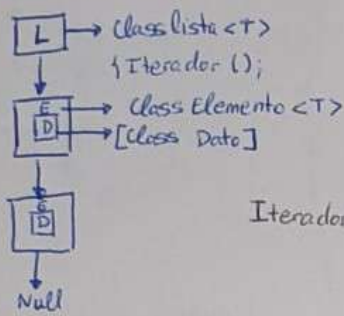


Listas



Si borro D1, todos se mueven y uso get(i)

Algunas veces es mejor usar diccionarios o listas

Iterador { Tiene interface <T> "da contrato a usar" HaySiguiente();
Guarda estado iteración de una lista GetDato();

↳ almacena datos no estados

Class Iterador {

// Constructor

Iterador (lista l) { }

HaySiguiente () { }

GetDato () { }

Lista <T> milista;

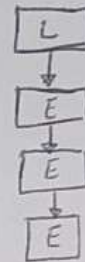
Elemento <T> actual;

Estado

Iterador (lista <T> l) {

this.milista = l;

this.actual = l.primerElemento; }



boolean haySiguiente () {
return this.actual != null; }

T getDato () {

T temporal = this.actual.getDato ();

this.actual = this.actual.Siguiente ();

return temporal; }



class Lista <T> {

Iterador () {

return new Iterador <T> (this);

static void main () {

lista <String> l = new lista <String> ();

l.add ("Hola")

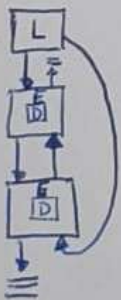
l.add ("Mundo!")

IIterador <String> i = l.Iterador ();

while (i.haySiguiente())

System.out.println (i.getDato ()); }

Doblemente enlazada



Class ListaDE {

ElementoDE primero

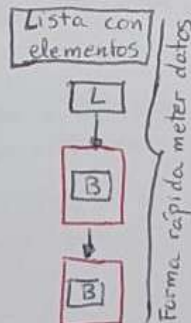
ElementoDE ultimo {

Class Elemento DE {

T dato;

ElementoDE anterior;

ElementoDE siguiente; }



Forma rápida meter datos

Int add (x nulo) {

Elemento SE <x> temporal =
new Elemento <x> (nulo);

if (getPrimerElemento () == null)

while (flotante.getSiguiente () < null) {

flot

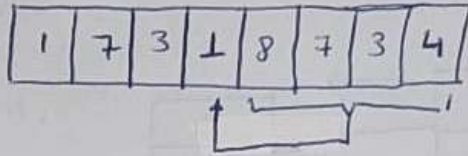
Listas

Listas simplemente enlazadas

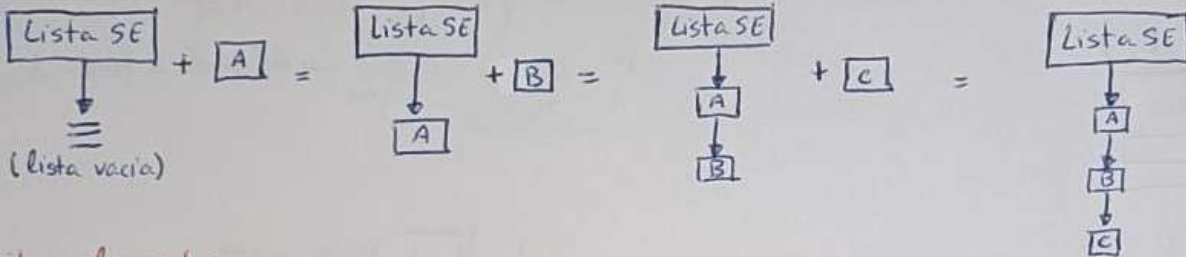
```
class Lista < tipoDato >
    Lista < Integer > a = new Lista < Integer > ()
```

```
tipoDato datos [] = new tipoDato () [7]
```

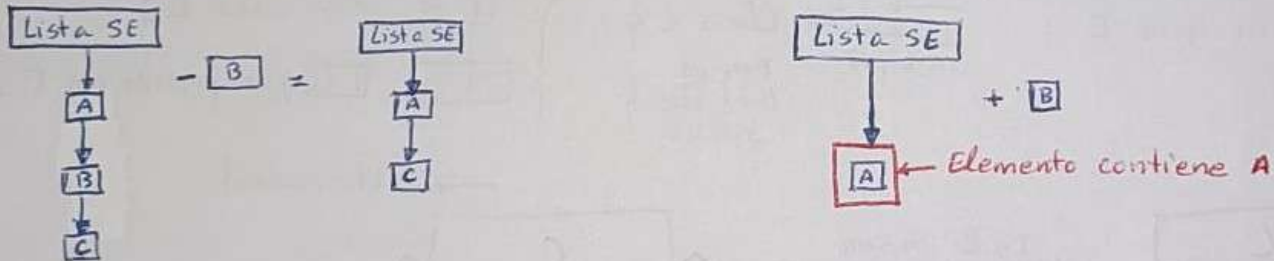
Usamos Listas dinámicas



Añadir elementos



Quitar elementos



```
class Lista < x > {
```

~~x~~ { x primer elemento;
x segundo elemento; }

✓ { Elemento SE < x > primer Elemento; }

// Constructor

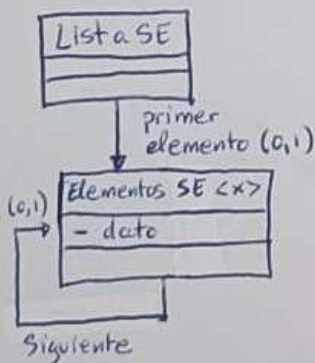
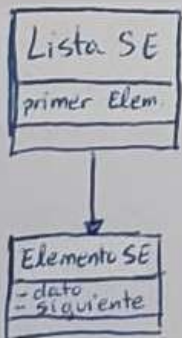
```
class Elemento SE < x > {
```

x dato;

Elemento SE < x > siguiente; }

Atributos

↳ Es de este tipo
Está declarado



```
Lista SE < Integer > milista = new Lista SE < Integer > ();
```

...

```
void miPrueba () {
```

```
Integer a = new Integer (5);
```

```
milista.add(a); }
```

Añadimos
valor en el
dato directamente

Ej: añadir { Elemento SE (x mid to)
this.dato = mid to

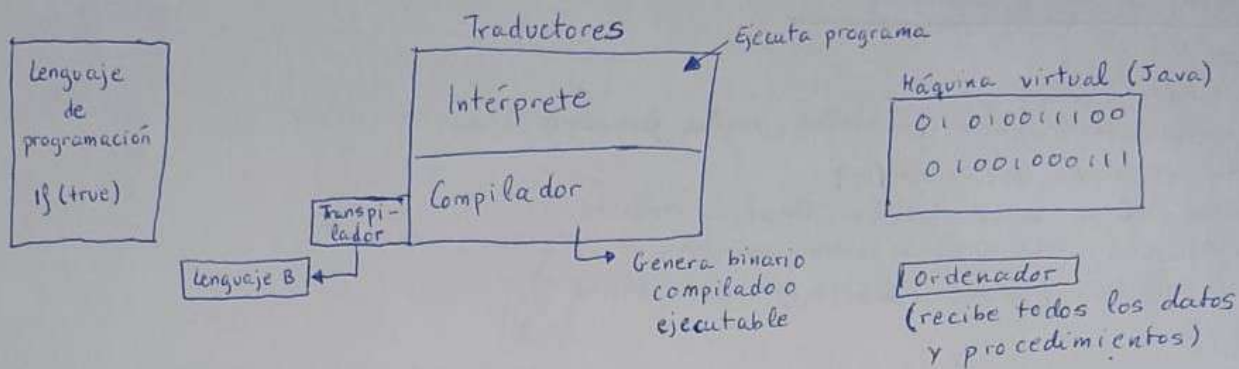
```
Probar elemento SE () { }
```

```
int add (x dato) {
```

```
Elemento SE < x > temporal = new Elemento < x > (dato)
```

```
temporal.setSiguiente (this.primer Elemento);
```

```
setPrimer Elemento (temporal); }
```

Proceso

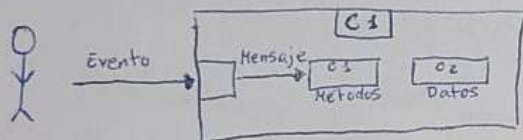
- Abstracción
- Reutilización
- Encapsulamiento → principio ocultación
- Modularidad
- Herencia
- Polimorfismo
- Recolección basura

Clase

- Objeto → Implementación
- Métodos → Funcionalidad de clases
 - ↳ Invocación (mensaje que envía los métodos)
- Evento → mensaje externo (no controla receptor)
- Atributo / propiedad → dato, variable de clase, ...
- Identificador / referencia
- Estado interno

Diagramas

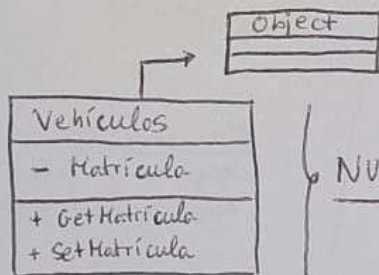
- De uso (interacciones usuario y sistema)
- Clases (representa clases, atributos, métodos y relaciones)
- Secuencia (orden e interacción de objetos)
- Estado (estados y transiciones de un objeto)
- Componentes (organización y dependencia hardware)



Clases abstractas

Nombre clase
Dato 1 (Privado) -
Dato 2 (Protegido) #
Dato 3 (Público) +
Método 1
Método 2
Método 3

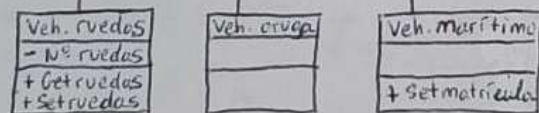
Ej



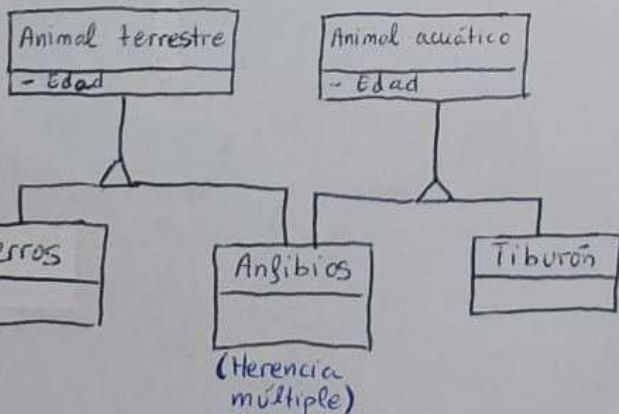
NUNCA PUEDES INSTANCIARLO (NEW)

(Reutilizar código) → Establecemos jerarquía de herencia

Todos tienen
SetMatrícula



(matrícula ≠ vehículo) → @override




(Herencia múltiple)

- La herencia múltiple en Java está prohibida
- Para ello usamos interfaces, con el fin de reutilizar código



Forma más rápida de meter datos en listas

• Buscamos caso especial

• Lista vacía:  no poder acceder a datos

int add(x nulo);

ElementoSE <x> temporal = new Elemento <x> (nulo);

① if (getPrimerElemento () == null)
setPrimerElemento (temporal);

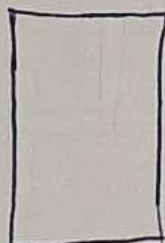
② else

ElementoSE <x> flotante = getPrimerElemento ();
while (flotante.getSiguiente () <> null) {
flotante = flotante.getSiguiente ();
flotante.setSiguiente (temporal); }

ListaSE <x>
n elemento:
add (x)
get NElementos ();
int NElementos ();

Directamente

Diccionarios



Lista <K> getKey ();
Iterador <K,v> it = this.getIterador ();
Lista <K> closes = new Lista <K> ();
while (it.hasNext ()) {
it.next ();
closes.add (it.getKey ()); }
return closes ; }

Iterador getIterador ();

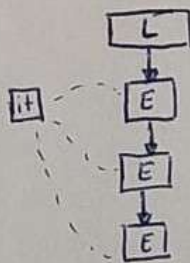
• Iterador D find (K); //reutilizar código { si solo hay una instrucción, no hace falta claves

• Iterador D find (Iterador D, K);

boolean exists (K clave) {
Iterador <K,v> it = this.getIterador ();
while (it.hasNext ()) {
it.next ();
if (it.getKey () == clave)
return true ; }
return false ; }

GetValue (K clave) {
Iterador <K,v> it = new Iterador ();
while (it.hasNext ()) {
it.next ();
if (it.getKey () == clave)
return it.getValue ();
return null ; }

Iterador D <K,v> find (Iterador D <K,v> it, K clave) {
while (it.hasNext ()) {
it.next ();
if (it.getKey () == K)
return it ; }
return { -null
-it }



Protegida

Iterador D <K,v> find (K clave) {
Iterador D <K,v> it = new Iterador D ();
return (this.find (it, clave)); }
getValue (K clave) {
return this.find (clave).getValue (); }
boolean insert (K clave, V dato) {
Iterador D <K,v> it = this.find (clave)
if (it.getActual () != null)
it.getActual ().setValue (dato);
else
this.add (clave, dato); }

boolean exists (K clave)

Uno { - return this.find (clave).hasNext ();
- return this.find (clave).getKey () != null ; }

ElementoD <K,V>

ElementoD <K,V> anterior
ElementoD <K,V> siguiente

K Indice;

V Dato;

boolean delete ()

K getKey ();

V getValue ();

boolean updateValue ()

IteradorD <K,V>

Diccionario <K,V> miDiccionario

ElementoD <K,V> actual

boolean hasNext ()

V next ()

K getKey () ← no avanza

V getValue () ← no avanza

Diccionario <K,V>

ElementoD <K,V> Primero

ElementoD <K,V> Ultimo

boolean add (K,V)

boolean insert (K,V)

boolean delete (K)

Lista <K> getKeys ()

Lista <V> getValues ()

boolean exists (K)

V getValue (K)

boolean setValue (K,V)