

## Resumen PAV (7)

*Este resumen ha sido elaborado para ser leído después de haber asistido a clase: pretende fijar y profundizar en los conceptos ya explicados.*

# Listas y *hashes* (I)

## Introducción

Los programas necesitan crear objetos. Dependiendo del problema que estemos resolviendo, en ocasiones será posible conocer de antemano el número de objetos que necesitará manejar el programa. Pero otras veces no podremos saber de antemano cuántos objetos necesitaremos crear.

Java tiene muchos modos de almacenar objetos. Además de poder almacenar la referencia a un objeto directamente en una variable, el lenguaje implementa arreglos (*arrays*). Una vez creado el arreglo, se puede acceder a sus elementos individuales usando un índice. Suponiendo que existiera una clase `Racional`, podríamos escribir:

```
// espacio para 200 objetos de tipo Racional
Racional[] racionales = new Racional[200];

// creo un objeto de tipo Racional, y lo almaceno en el array
racionales[0] = new Racional(5, 4);
```

En contrapartida a su velocidad de acceso aleatorio, los arreglos tienen tamaño fijo, y una vez creados no pueden aumentar ni disminuir su capacidad.

Java, como la mayoría de lenguajes modernos, tiene librerías que permiten manejar colecciones de objetos (*Collections*). Una colección es una clase que, como un arreglo, maneja agrupaciones de objetos, pero su tamaño es flexible y, dependiendo de la colección de que se trate, optimizan el acceso, la iteración de los elementos, mantienen la colección ordenada según un criterio, etc.

Antes de entrar a estudiar la librería de colecciones de Java, explicaremos la teoría de las listas enlazadas y *hashes*. Las listas enlazadas y los *hashes* son un tema clásico de ciencias de la informática. A nosotros nos interesa saber los conceptos generales sobre cómo funcionan y cómo se implementan (por lo menos a nivel básico) para poder entender mejor cómo funciona la librería de colecciones de Java.

En los programas que desarrollemos, salvo necesidad especial, usaremos la librería de colecciones de Java (*Java Collections Framework*) para almacenar nuestros objetos. Como explican los *tutorials* de Java<sup>1</sup>, el uso de la librería de colecciones nos permite usar clases ya probadas, conocidas y optimizadas, y centrarnos en la solución del problema que tenemos entre manos, sin necesidad de reinventar la rueda cada vez.

---

<sup>1</sup> <http://java.sun.com/docs/books/tutorial/collections/intro/index.html>

## Listas enlazadas<sup>2</sup>

Una lista enlazada es una serie de objetos en la que cada objeto contiene dos tipos de información:

a) los atributos propios del objeto (las variables de la clase que representan el estado del objeto en cuestión);

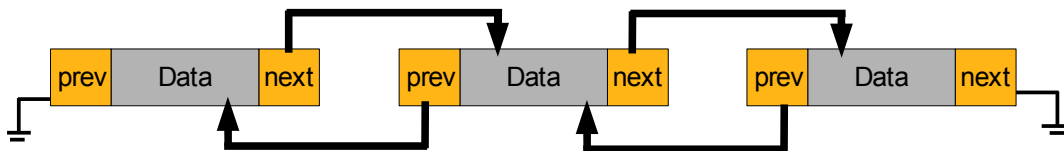
b) un atributo del tipo del mismo objeto, en el que se almacena la referencia a otro objeto de la lista. Es decir, una variable que apunta al siguiente elemento de la lista.

Los objetos que forman la lista se denominan **nodos**. Según cómo esté implementada la lista,

- cada nodo puede tener únicamente una referencia al siguiente nodo de la lista. Son las listas “simplemente” enlazadas, *single linked lists*. Sólo se pueden recorrer en una dirección (de inicio a fin);



- cada nodo puede tener dos referencias, una al nodo anterior y otra al nodo siguiente. Son listas “doblemente” enlazadas, *double linked lists*, que pueden recorrerse tanto hacia adelante como hacia atrás).



Las listas permiten iterar rápidamente sus elementos (acceso secuencial), e insertar o eliminar un elemento en tiempo constante y con un mínimo de operaciones. En cambio acceder a un elemento determinado de la lista (acceso aleatorio) es una operación costosa en ciclos de máquina.

Otros tipos de estructuras (*Stacks*, *Queues*) utilizan listas enlazadas en su implementación.

### Implementación de una lista simplemente enlazada en Java

Veamos como podrían declararse estas clases en Java para una lista simplemente enlazada. Se suele usar `null` en la referencia del último nodo de la lista, para indicar que la lista ha terminado. (El caso de la lista doble es similar.)

Nodo.java

```
public class Nodo
{
    private Object data;
    private Nodo next;

    // constructor
    public Nodo(Object data) { this.data = data; }
```

<sup>2</sup> Se puede encontrar información adicional en los apuntes del curso en la intranet y en Wikipedia: [http://en.wikipedia.org/wiki/Linked\\_lists](http://en.wikipedia.org/wiki/Linked_lists)

```

    public Nodo getNext()          { return next;      }
    public void setNext(Nodo next) { this.next = next; }

    public Object getData()        { return data;      }
    public void setData(Object data){ this.data = data; }

    /**
     * inserta un nodo delante del este nodo
     */
    public void insertNodo(Object data) {
        Nodo nuevo = new Nodo(data);
        nuevo.setNext(next);
        next = nuevo;
    }

    /**
     * borra el nodo siguiente
     */
    public void deleteNodo() {
        // antes de borrar, veo si no estoy al final de la lista
        if(next != null) {
            next = next.getNext();
        } // end if
    }
}

```

La variable `data` es de tipo `Object`. Esto nos permite usar la lista para almacenar cualquier tipo de objetos, pero obliga a usar un *downcast* cada vez que se use una funcionalidad específica de la clase del objeto almacenado en el nodo.

Ahora escribiremos un programa que use la clase `Nodo`. Necesitamos algún tipo de objeto para almacenar en la lista. Para eso, usaremos la clase `Persona`, que definimos a continuación:

```

Persona.java
public class Persona
{
    private String nombre;
    private String apellido;
    private int edad;

    // Constructor
    public Persona(String nombre, String apellido, int edad) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }

    public String getNombre() { return nombre; }
    public String getApellido() { return apellido; }
    public int getEdad() { return edad; }

    // override del método toString()
    public String toString() {
        return apellido + ", " + nombre + " (" + edad + " años)";
    }
}

```

A continuación una clase que usa `Nodo` para guardar `Personas`:

#### TestNodo.java

```
public class TestNodo
{
    public static void main(String[] args) {
        Nodo lista = null;

        // insertamos algunos objetos Persona en la lista
        // el primer nodo lo asignamos directamente a lista

        lista = new Nodo(new Persona("Juan","Pérez", 45));

        // el resto de nodos son insertados en la segunda posición

        lista.insertNodo(new Persona("José", "Pérez", 30));
        lista.insertNodo(new Persona("Juana", "Pérez", 28));
        lista.insertNodo(new Persona("Benjamín", "Pérez", 18));

        // imprimimos la lista
        System.out.println("-----");
        printList(lista);

        // ahora vamos a borrar el *segundo* nodo:
        lista.deleteNodo();
        System.out.println("-----");
        printList(lista);

        // para acceder a los métodos de Persona, necesitamos
        // hacer un downcast (salvo que sean override de los
        // métodos de Object, en cuyo caso la llamada es polimórfica)
        // recuperamos la persona del segundo elemento de la lista

        Persona p = (Persona) lista.getNext().getData();
        System.out.println("Apellido: " + p.getApellido());
    }

    /**
     * imprime la lista desde el nodo list en adelante
     */
    public static void printList(Nodo list) {
        while(list != null) {
            // imprimimos el nodo.  getData devuelve un
            // Object, pero no necesitamos hacer un downcast:
            // toString (que es llamado por el println)
            // está definido en Object, Persona hace
            // un override de toString.  Por tanto, la llamada al toString
            // de Object es polimórfica.

            System.out.println( list.getData() );

            // avanzamos al siguiente nodo
            list = list.getNext();
        } // end while
    }
}
```

### ***Generics para especificar el tipo de dato que guarda la lista***

Releyendo la clase `Nodo` que hemos escrito, podríamos preguntarnos: ¿no sería posible usar un *generic* para que nuestro `Nodo` sepa qué tipo de objeto está almacenando? Así nos podríamos ahorrar el *downcast* cada vez que necesitemos acceder a los métodos de `Persona`.

Definir una clase que luego permita a los que la usan especificar un tipo de datos usando *generics* no es complicado. Por ejemplo, si definimos una clase `Test` así:

```
class Test<TipoDeDato> {
    TipoDeDato[] arreglo = new TipoDeDato[100];

    public TipoDeDato get(int i) {
        return arreglo[i];
    }
    public void set(int i, TipoDeDato dato) {
        arreglo[i] = dato;
    }
}
```

podremos crear objetos de `Test` que guardan hasta 100 objetos de cualquier clase que indiquemos, sin tener que preocuparnos del *downcast*:

```
Test<Persona> lista = new Test<Persona>();
lista.set(0, new Persona("Juan", "Pérez", 45));
// en vez de ((Persona) lista).get(0)
System.out.println(lista.get(0));
```

A efectos del *generic*, podemos imaginar que cuando se ejecuta el programa, `TipoDeDato` es reemplazado en la clase `Test` por el tipo de dato que haya especificado el usuario entre brackets (< >) al declarar que una variable es de tipo `Test`.

Cambiamos la clase `Nodo` para que el tipo de objeto almacenado en el nodo se especifique usando *generics*:

NodoGenerics.java

```
public class NodoGenerics<Tipo> {

    // aquí era private Object data
    private Tipo data;
    private NodoGenerics<Tipo> next;

    public NodoGenerics(Tipo data) {
        this.data = data;
    }

    public NodoGenerics<Tipo> getNext() { return next; }
    public void setNext(NodoGenerics<Tipo> next) { this.next = next; }

    // aquí era public Object getData
    public Tipo getData() { return data; }
    public void setData(Tipo data) { this.data = data; }

    /**
     * inserta un nodo delante del este nodo
     */
    public void insertNodo(Tipo data) {
        NodoGenerics<Tipo> nuevo = new NodoGenerics<Tipo>(data);
        nuevo.setNext(next);
        next = nuevo;
    }

    /**
```

```

        * borra el nodo siguiente
        */
    public void deleteNodo() {
        if(next != null) {
            next = next.getNext();
        }
    }
}

```

#### TestNodoGenerics.java

```

public class TestGenericsNode
{
    public static void main(String[] args) {
        // declaro el nodo usando generics: un nodo
        // que almacena objetos Persona
        NodoGenerics<Persona> lista = null;

        lista = new NodoGenerics<Persona>(new Persona("Juan","Pérez", 45));
        // el resto de nodos son insertados en la segunda posición
        lista.insertNodo(new Persona("José", "Pérez", 30));
        lista.insertNodo(new Persona("Juana", "Pérez", 28));
        lista.insertNodo(new Persona("Benjamín", "Pérez", 18));

        // imprimimos la lista
        System.out.println("-----");
        printList(lista);

        // ahora, vamos a borrar el *segundo* nodo:
        lista.deleteNodo();
        System.out.println("-----");
        printList(lista);

        // aquí ya no necesito hacer el downcast
        Persona p = lista.getNext().getData();
        System.out.println("Apellido: " + p.getApellido());
    }

    /**
     * imprime la lista desde el nodo list en adelante
     */
    public static void printList(NodoGenerics<Persona> list) {
        while(list != null) {
            System.out.println( list.getData() );
            list = list.getNext();
        } // end while
    }
}

```

## **Otras estructuras de datos que usan listas enlazadas**

### a) Pilas (stacks)

Un *stack* es una estructura de datos que sólo permite insertar y quitar elementos en uno de los extremos. Dicho de otro modo, se retira el elemento que haya sido insertado más recientemente. Por eso se dice que es una estructura LIFO (*Last-in, First-out*): el último elemento insertado es el primero en ser recuperado.

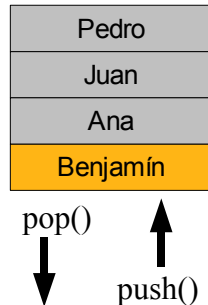
Los nombres de los métodos de un *stack* suelen ser bastante estándares:

void push(Object o)	inserta el objeto o en el stack
Object pop()	retira el último elemento insertado del stack

Object peek()

el último elemento insertado, pero sin quitarlo del stack

Además suele haber un método empty() o isEmpty() que devuelve un boolean true si el stack está vacío.



A continuación presentamos la implementación de un *stack* usando *generics*, y una clase que ejemplifica su uso:

```
Stack.java
public class Stack<Tipo>
{
    private StackNode<Tipo> head = null;

    /**
     * inserta un elemento en el stack
     */
    public void push(Tipo dato) {
        // crea un nuevo nodo y lo inserta delante del primero
        StackNode<Tipo> nodo = new StackNode<Tipo>(dato);
        nodo.setNext(head);
        head = nodo;
    }

    /**
     * pop - devuelve el último elemento insertado y lo quita
     * del stack
     */
    public Tipo pop() {
        if(!isEmpty()) {
            StackNode<Tipo> node = head;
            head = head.getNext();
            return node.getData();
        } else
            return null;
    }

    // end pop

    /**
     * peek - devuelve el último elemento insertado, pero
     * sin quitarlo de la pila
     */
    public Tipo peek() {
        if(!isEmpty()) {
            return head.getData();
        } else
            return null;
    }

    // end peek

    /**
     * devuelve true si el stack está vacío
     */
}
```

```

        */
        public boolean isEmpty() {
            return head == null;
        } // end isEmpty
    }

    class StackNode<Tipo> {
        private Tipo data;
        private StackNode<Tipo> next;

        public StackNode(Tipo data) {
            this.data = data;
        }

        public void setNext(StackNode<Tipo> next) {
            this.next = next;
        }

        public StackNode<Tipo> getNext() {
            return next;
        }

        public void setData(Tipo data) {
            this.data = data;
        }

        public Tipo getData() {
            return data;
        }
    }
}

TestStack.java
public class TestStack
{
    public static void main(String[] args) {

        Stack<Persona> stack = new Stack<Persona>();

        stack.push(new Persona("Juan", "Pérez", 45));
        stack.push(new Persona("José", "Pérez", 30));
        stack.push(new Persona("Juana", "Pérez", 28));
        stack.push(new Persona("Benjamín", "Pérez", 18));

        System.out.println("Peek: " + stack.peek());

        while(!stack.isEmpty()) {
            System.out.println("pop: " + stack.pop());
        }
    }
}

```

Se pueden implementar también stacks de tipo FIFO (*first-in, first-out*): los objetos se insertan por abajo de la pila y se retiran por arriba, de modo que el primero en insertarse es el primero en salir.

## Hashes

### Qué es un hash

Un *hash* es un número que se usa para representa a un objeto. Una *función hash* es una función que recibe como parámetro un objeto y devuelve un número tal que se cumple:

- para el mismo objeto, la función siempre devuelve el mismo número hash.



- si dos objetos son iguales, entonces deben tener el mismo número hash.
- si dos objetos son distintos, en la medida de lo posible, no tienen el mismo número hash<sup>3</sup>. Si dos objetos distintos tienen el mismo número *hash*, se dice que hay una **colisión**.

## ***Para qué sirve un hash***

Los arreglos permiten acceder a los elementos que almacenan de modo rápido y preciso usando un entero como índice:

```
Persona[] lista = new Persona[100]; // una lista de 100 personas
(...)
Persona juan = lista[15];
```

Sin embargo hay otras situaciones en las que necesitamos acceder a un elemento del arreglo usando como índice un tipo de dato distinto a un número. Por ejemplo, consideremos una clase que implementa una lista de palabras y sus definiciones para un diccionario. Cada “entrada” del diccionario consiste en la palabra y su definición:

```
class Entrada {
    String palabra;
    String definicion;
}
```

Supongamos que nuestro diccionario está ya creado y almacenado en un arreglo *diccionario*. Si queremos buscar la definición de la palabra “perro” en el diccionario, no podemos acceder directamente al elemento del diccionario que contiene la definición, porque no sabemos en qué posición de *diccionario* está. Estamos forzados a usar algún método de búsqueda para iterar los objetos *Entrada* de *diccionario* hasta encontrar el objeto en el que el atributo *palabra* es igual a *perro*:

```
String searchString = “perro”;
String definicion = null;

for(int i = 0; i < diccionario.length; i++) {
    if (diccionario[i].palabra.equals(searchString)) {
        definicion = diccionario[i].definicion;
        break;
    }
}
```

En realidad, lo que desearíamos es poder acceder a los elementos de *diccionario* usando la palabra que buscamos como índice:

```
String definicionDePerro = diccionario[“perro”];
```

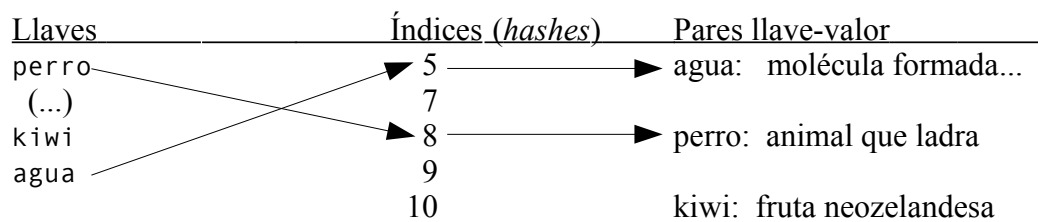
Otros lenguajes (por ejemplo Python, Ruby, PHP) tienen un tipo de datos que usa un *String* como índice del arreglo: este tipo de datos se llama *diccionario*, o también *arrays asociativo*. Sin embargo, en Java los arrays asociativos no son parte del lenguaje<sup>4</sup>.

---

<sup>3</sup> También se habla de funciones hash en criptografía. En ese contexto, las funciones hash tienen requisitos adicionales que no son materia de este resumen. Se puede consultar, por ejemplo, BRUCE SCHNEIDER, *Applied Cryptography*.

<sup>4</sup> Se implementan usando una serie de clases de la librería.

La solución a este problema es usar una tabla de hash: usar una función hash para convertir la **llave** (*key*) en un número entero; una vez que tenemos el número entero, podemos usarlo como índice de un arreglo convencional para acceder a la data (el valor asociado).



En teoría, con la función *hash* adecuada podríamos escribir algo así:

```
Entrada[] diccionario = new Entrada[1000];
/*
  (...) aquí lleno de algún modo el diccionario con las definiciones
*/
```

```
String definicionDePerro = diccionario(hashCode("perro"));
```

Sin embargo, la cosa no es tan sencilla. Para implementar una buena tabla hash, se necesita:

a) Un buena función hash. Para que el hash pueda usarse como índice de un arreglo de una tabla hash, es necesario que el rango de valores que devuelve la función hash esté dentro de un rango razonable. Si quiero implementar un diccionario de 1000 palabras, y el hash de "perro" es 1.543.756, no sería lógico reservar espacio para un arreglo de millón y medio de elementos.

Por eso, muchas veces el valor que devuelve la función hash es convertido en un número que esté dentro del rango de índices del array (pero esto aumenta la probabilidad de una colisión). Una técnica común es tomar el residuo de la división entera del resultado de la función hash entre el número de posiciones de la tabla (se llama método de división).

La siguiente clase contiene dos funciones: *RSHash*, que genera un hash para un objeto de tipo *String*; y *remapHash*, que remapea un valor de modo que esté dentro de las dimensiones de la tabla (indicada por la variable *tamanoTabla*).

```
public class Hash {

    int tamanoTabla = 1023;

    // función hash (función "mágica")
    public long RSHash(String str) {
        int b = 378551;
        int a = 63689;
        long hash = 0;
        for(int i = 0; i < str.length(); i++) {
            hash = hash * a + str.charAt(i);
            a = a * b;
        }
        return (hash & 0x7FFFFFFF);
    } // end RSHash

    public long remapHash(long hash) {
```

```

        return hash % tamanoTabla;
    } // end remapHash

    public Hash() {
        System.out.println("Hashes -----");
        System.out.println("Hash para 'perro': " + RSHash("perro"));
        System.out.println("Hash para 'gato': " + RSHash("gato"));

        System.out.println("\nHashes remapeados para a una tabla de "
            + tamanoTabla + " elementos");
        System.out.println("-----");
        System.out.println("Hash para 'perro': "
            + remapHash(RSHash("perro")));
        System.out.println("Hash para 'gato': "
            + remapHash(RSHash("gato")));
    }

    public static void main(String[] args) {
        new Hash();
    }
}

```

Es deseable también que los números que produce la función hash estén distribuidos de modo uniforme dentro del rango. Cuando parte de los resultados de la función hash se acumulan alrededor de pocos valores, se dice que hay *clustering*.

Google Code tiene una página interesante con algunas de las funciones de hash que usa Google en su buscador: <http://goog-sparsehash.sourceforge.net/>. En todo caso, el tema de cómo diseñar una función hash está más allá de los objetivos del curso.

b) Un manejo adecuado de las colisiones. Si se produce una colisión<sup>5</sup>, la implementación de la tabla hash debe solucionarla de algún modo.

Dos técnicas populares de resolución de colisiones son:

- hash de encadenamiento (*chaining*) Se crea en cada dirección de la tabla hash una lista enlazada (“lista encadenada”). Cuando más de un objeto distinto tiene el mismo hash, se guarda en la lista enlazada que está en esa posición de la tabla hash.
- el hash direccionadamente abierto (*open addressing*) Cuando se produce una colisión, no se usa una lista enlazada, sino que se busca una posición libre en el arreglo para guardar el valor. Esta técnica requiere una tabla de mayor tamaño que los valores que se quieren almacenar.

Hay diversas estrategias para encontrar el siguiente valor libre: por ejemplo, búsqueda lineal (*linear probing*), cuadrática (*quadratic probing*), etc.

## ***El método hashCode de Object***

La clase Object de Java define el método `public int hashCode()`. Las clases definidas

<sup>5</sup> De la voz hash\_table en Wikipedia ([http://en.wikipedia.org/wiki/Hash\\_table](http://en.wikipedia.org/wiki/Hash_table)): “To give an idea of the importance of a good collision resolution strategy, consider the following result, derived using the [birthday paradox](#). Even if we assume that our hash function outputs random indices [uniformly distributed](#) over the array, and even for an array with 1 million entries, there is a 95% chance of at least one collision occurring before it contains 2500 records.”

La paradoja del cumpleaños dice que si hay más de 23 personas en una habitación, la probabilidad de que dos tengan cumplan años el mismo día es más del 50%.

en la librería de Java (por ejemplo, `String`) ya tienen un método `hashCode` adecuado. En cambio, si necesitamos usar como llave (*key*) nuestros propios objetos, entonces será necesario hacer un *override* del método `hashCode`.

La documentación de la librería de Java especifica qué condiciones tiene que cumplir el método `hashCode`, que son similares a las que ya hemos considerado antes:

- Si `hashCode` es invocado más de una vez sobre el mismo objeto durante la ejecución del programa, y el estado del objeto no ha cambiado, `hashCode` debe devolver siempre el mismo número entero. Se considera que el estado del objeto ha cambiado cuando la información que se usa en una comparación usando el método `equals` del objeto ha cambiado (en otras palabras, basta que no cambien las variables que se usan en la comparación). En cambio, no hace falta que el entero devuelto sea el mismo cada vez que se ejecuta el programa.
- Si dos objetos se consideran iguales usando el método `equals`, entonces la llamada de `hashCode` sobre cada uno de los objetos debe dar el mismo resultado.
- No es necesario que los dos objetos distintos produzcan resultados distintos.

Es conveniente revisar los apuntes del curso<sup>6</sup> que están en la intranet para ver los modos más usuales de manejar las colisiones y de implementar las tablas de hashes. También se puede encontrar un buen resumen en la voz *Hash table* de Wikipedia ([http://en.wikipedia.org/wiki/Hash\\_table](http://en.wikipedia.org/wiki/Hash_table)).

### ***Entonces, ¿para qué necesitamos saber cómo funciona una función hash?***

Algunas de las clases de la librería de colecciones de Java implementan tablas de hash. Si en esas colecciones usamos como llave un tipo de objeto que ya tenga un método `hashCode` implementado, entonces no necesitamos hacer un *override* de `hashCode`.

En cambio, si usamos como llave del hash un objeto que no tiene `hashCode` implementado<sup>7</sup>, **entonces necesitamos hacer un *override* del método `hashCode`**, porque de lo contrario la colección no funcionará como esperamos (no podremos recuperar los objetos).

Comentarios, correcciones y sugerencias: Roberto Zoia ([roberto.zoia@gmail.com](mailto:roberto.zoia@gmail.com))

This work is licensed under the Creative Commons Attribution-ShareAlike 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

---

<sup>6</sup> GUEVARA ÁLVAREZ, ERNESTO. *Apuntes de Programación Avanzada*. Facultad de Ingeniería, Universidad de Piura. Enero 2006.

<sup>7</sup> Por ejemplo, una clase que nosotros hayamos definido.