

Conceptos de la Informática

(JAVA)

Glosario

Datos: elementos que van a proporcionar la entrada de información de los programas que se podrá almacenar y procesar.

Algoritmo: secuencia de instrucciones bien implementadas y ordenadas que resuelve una tarea en concreto.

Programa: totalidad de las instrucciones y algoritmos escritos mediante un lenguaje de programación.

Anidamiento: definir bloques dentro de otros bloques.

Identificadores: nombres que sirven para identificar los elementos del programa, como por ejemplo el nombre de una variable, una función, una clase. Restricciones:

- letras sin contar la ñ (minúsculas o mayúsculas), dígitos del 0 al 9, guion bajo (_) y símbolo del dólar (\$). No se admiten espacios en blanco ni acentuaciones.
- Deberán empezar forzosamente por una letra, mayúscula o minúscula.
- No se podrán repetir los nombres de los identificadores.
- No se podrán utilizar palabras reservadas del lenguaje.
- Los nombres deberán ser significativos, es decir, que den a entender qué representará ese identificador.

Variable: posición en memoria al que vamos a dar un nombre. Ésta tendrá un valor que podrá cambiar

a lo largo del programa. Tienen tres partes:

- El tipo.
- El nombre.
- El valor.

Constante: tipo de variable, pero cuyo valor, una vez asignado, no va a poder cambiar durante la ejecución del programa. Utilizaremos la palabra reservada `final` para declarar constante cualquier tipo de dato.

Sentencias: son líneas de código que componen un programa y que se ejecutan secuencialmente.

IDE (Entorno de desarrollo integrado)

Los IDE incorporan los siguientes componentes:

- **Editores de texto:** Para poder escribir el código fuente.
- **Preprocesadores:** Este componente preprocesa el código fuente para poder facilitarle el trabajo al compilador.
- **Ensambladores:** Estos son los encargados de generar código ensamblador si es necesario.
- **Linkers:** Son los encargados de enlazar las bibliotecas externas utilizadas.
- **Depuradores:** Se encargan de la depuración del código fuente para la detección y eliminación de errores en tiempo de ejecución.

Hay muchos tipos de IDE, podemos clasificarlos en:

- IDE libres.
- IDE privativos.
- IDE multiplataforma.
- IDE monoplataforma.

El proceso de compilación pasa por las siguientes fases:

- Análisis léxico, sintáctico y semántico.
- Generación de código intermedio.
- Optimización de código.
- Generación de tabla de símbolos.

Paradigmas de programación

Indica la forma en la que debe usarse un lenguaje y la finalidad y comportamiento de este,

- **Programación estructurada:** Este es el paradigma más simple. En este las instrucciones se ejecutan en el orden en el que se escriben y siempre se ejecutarán en el orden.
- **Programación funcional:** Es una mejora de la programación estructurada en la que el código se agrupa en bloques llamados funciones, que pueden ser invocadas tantas veces como sea necesario, lo cual permite reutilizar el código sin tener que volverlo a escribir. Reduce considerablemente la cantidad de líneas de código de los programas, es decir, aumenta la reusabilidad de código.

- Programación orientada a objetos: Este paradigma agregó el concepto de clase y objeto, así como las relaciones de herencia y el polimorfismo. Permite agrupar los datos en plantillas denominadas clases, de las que luego se crearán copias (los objetos) con los valores concretos. Ayuda también a la reutilización de código de los programas.

- Programación orientada a eventos: En este paradigma, la ejecución de código viene dada por los eventos que ocurren en el sistema.

Tipos de datos enteros

Tipo	Bytes ocupados en memoria	Rango de valores
byte	1	$[-128, 127]$
short	2	$[-32768, 32767]$
int	4	$[-2^{31}, 2^{31}-1]$
long	8	$[-2^{63}, 2^{63}-1]$

Tipos de datos reales

Tipo	Bytes ocupados en memoria	Rango de valores	
		En los negativos	En los positivos
float	4	$[-3.4E^{38}, -1.4E^{45}]$	$[1.4E^{-45}, 3.4E^{48}]$
double	8	$[-1.8E^{308}, -4.9E^{324}]$	$[4.9E^{-324}, 1.48E^{308}]$

Tipos de datos booleanos

Tipo	Bytes ocupados en memoria	Rango de valores
boolean	8	true / false

Expresiones y operadores aritméticos

Operación	Símbolo	Descripción
Suma	+	Suma dos números
Resta	-	Resta dos números
Producto	*	Multiplica dos números
División	/	Divide con decimales dos números
Resto de la división	%	Obtiene el resto de una división

Operadores lógicos

- Operador AND, representado como &&.
- Operador OR, representado como ||.
- Operador NOT, representado como !.

AND	TRUE	FALSE		OR	TRUE	FALSE		NOT	
TRUE	TRUE	FALSE		TRUE	TRUE	TRUE		TRUE	FALSE
FALSE	FALSE	FALSE		FALSE	TRUE	FALSE		FALSE	TRUE

Operadores relacionales y de asignación




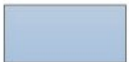

- Operador igual que, se representa como ==.
- Operador distinto, se representa como !=.
- Operador mayor que, se representa como >.
- Operador menor que, se representa como <.
- Operador mayor o igual que, se representa como >=.
- Operador menor o igual que, se representa como <=.

- `int a += b; int a = a + b;`

- `int a -= b; int a = a - b;`
- `int a *= b; int a = a * b;`
- `int a /= b; int a = a / b;`
- `int a %= a % b; int a = a % b;`

Diagramas de flujo

- Sentencias if: Nos permiten decidir si queremos ejecutar o no un fragmento de código o no dependiendo de si se cumple una cierta condición.
- Sentencias if-else: Similar al anterior, permiten ejecutar un bloque de código si se cumple una condición u otro si no se cumple.
- Sentencias switch(case, break): Nos permite ejecutar un bloque diferente de código para cada valor posible que pueda tomar una expresión.

Símbolo	Nombre	Función
	Inicio / Final	Representa el inicio y el final de un proceso
	Línea de Flujo	Indica el orden de la ejecución de las operaciones. La flecha indica la siguiente instrucción.
	Entrada / Salida	Representa la lectura de datos en la entrada y la impresión de datos en la salida
	Proceso	Representa cualquier tipo de operación
	Decisión	Nos permite analizar una situación, con base en los valores verdadero y falso

Bucle while: cuando el número de iteraciones sea desconocido porque dependa de una condición, por ejemplo, solicitar números al usuario hasta que sumen un valor determinado.

```
int contador = 1;
while( contador <= 50 )
{
    System.out.println("El número es: " + contador);
    contador++;
}
```

Bucle do-while: cuando se tenga que ejecutar al menos una vez, por ejemplo, solicitar números al usuario hasta que introduzca el 0. Puede que lo introduzca a la primera o a la cuarta vez, pero tendrá que introducir al menos uno.

```
int contador = 1;
do
{
    System.out.println("El número es: " + contador);
    contador++;
} while( contador <= 50 );
```

Bucle for: cuando tengamos un número de iteraciones conocidas, por ejemplo, sumar los 100 primeros números enteros.

```
int i;

for (i = 0; i <= 10; i++) //Sentencia For
{
    System.out.println(i); //instrucciones del bucle
}
```

Instrucciones goto, break, continue y return

- Instrucción goto: nos permite transferir el control de ejecución del programa a otro punto identificado por una etiqueta previamente creada.
- Instrucción break: es utilizada para salir de un condicional o de un bucle de forma abrupta, pudiendo provocar fallos en los programas por el hecho de usarla incorrectamente. En Java la podemos utilizar, pero solo debemos hacerlo en las instrucciones switch.
- Instrucción continue: se utiliza en los bucles y su funcionamiento consiste en hacer que el bucle pase a la siguiente vuelta directamente, aumentando así su contador.
- Instrucción return: se utiliza en los procedimientos que veremos más adelante, y su funcionamiento consiste en devolver el control de ejecución al punto donde se llamó al procedimiento.

Concepto de array

Área de memoria conjunta formada por muchas variables del mismo tipo.

```
tipo[] nombrevariable = new tipo[ tamaño ];
```

Métodos Ordenación

Método burbuja:

```
public static void burbuja(int[] A) {
    int i, j, aux;
    for (i = 0; i < A.length - 1; i++) {
        for (j = 0; j < A.length - i - 1; j++) {
            if (A[j + 1] < A[j]) {
                aux = A[j + 1];
                A[j + 1] = A[j];
                A[j] = aux;
            }
        }
    }
}
```

Método selección:

```
//método java de ordenación por selección
public static void seleccion(int A[]) {
    int i, j, menor, pos, tmp;
    for (i = 0; i < A.length - 1; i++) {
        menor = A[i];
        pos = i;
        for (j = i + 1; j < A.length; j++){
            if (A[j] < menor) {
                menor = A[j];
                pos = j;
            }
        }
        if (pos != i){
            tmp = A[i];
            A[i] = A[pos];
            A[pos] = tmp;
        }
    }
}
```

Método de inserción:

```
public static void insercionDirecta(int A[]){
    int p, j;
    int aux;
    for (p = 1; p < A.length; p++){ // desde el segundo elemento hasta
        aux = A[p];                // el final, guardamos el elemento y
        j = p - 1;                // empezamos a comprobar con el anterior
        while ((j >= 0) && (aux < A[j])){ // mientras queden posiciones y el
                                        // valor de aux sea menor que los
                                        A[j + 1] = A[j]; // de la izquierda, se desplaza a
                                        j--;              // la derecha
        }
        A[j + 1] = aux;            // colocamos aux en su sitio
    }
}
```

Búsqueda lineal:

```
/**
 * Método que aplica la búsqueda lineal a un array
 *
 * @param array Array para buscar un elemento
 * @param elemento Elemento que se busca en el array
 * @return Devuelve true si el elemento está en el array, falso sino
 */
public static boolean busquedaLineal(int array[], int elemento) {
    boolean encontrado = false;

    for (int i = 0; i < array.length && !encontrado; i++) {
        if (array[i] == elemento) {
            encontrado = true;
        }
    }

    return encontrado;
}

public static void main(String[] args) {
    int[] array = new int[10]; // Declaración de un array
    // Relleno el array
    for (int i = 0; i < array.length; i++) {
        array[i] = i + 1;
    }

    // Busco el 4 en el array
    boolean estaEl4 = busquedaLineal(array, 4);
}
```


Búsqueda binaria:

```
/**
 * Método que aplica la búsqueda binaria a un array
 *
 * @param array Array para buscar un elemento
 * @param elemento Elemento que se busca en el array
 * @return Devuelve true si el elemento está en el array, falso sino
 */
public static boolean busquedaBinaria(int array[], int elemento) {
    boolean encontrado = false;
    int inicio = 0;
    int fin = array.length - 1;
    int posicion_buscar;

    while (inicio <= fin && !encontrado) {
        posicion_buscar = (inicio + fin) / 2;
        if (array[posicion_buscar] == elemento) {
            encontrado = true;
        } else {
            if (elemento > array[posicion_buscar]) {
                inicio = posicion_buscar + 1;
            } else {
                fin = posicion_buscar - 1;
            }
        }
    }

    return encontrado;
}
```

Arrays multidimensionales

Inicializar:

```
tipo[][] nombrevARIABLE = new tipo[filas][columnas]
```

Asignar valor:

```
variable[fila][columna] = valor
```

Operaciones con cadenas de caracteres

Método	Descripción
length()	Devuelve la longitud de la cadena
indexOf('caracter')	Devuelve la posición de la primera aparición de carácter
lastIndexOf('caracter')	Devuelve la posición de la última aparición de carácter
charAt(n)	Devuelve el carácter que está en la posición n
substring(n1,n2)	Devuelve la subcadena comprendida entre las posiciones n1 y n2-1
toUpperCase()	Devuelve la cadena convertida a mayúsculas
toLowerCase()	Devuelve la cadena convertida a minúsculas
equals("cad")	Compara dos cadenas y devuelve true si son iguales
equalsIgnoreCase("cad")	Igual que equals pero sin considerar mayúsculas y minúsculas
compareTo(OtroString)	Devuelve 0 si las dos cadenas son iguales. <0 si la primera es alfabéticamente menor que la segunda ó >0 si la primera es alfabéticamente mayor que la segunda.
compareToIgnoreCase(OtroString)	Igual que compareTo pero sin considerar mayúsculas y minúsculas.
valueOf(N)	Convierte el valor N a String. N puede ser de cualquier tipo.
replace (char viejoChar, char nuevoChar)	Reemplaza en el string que invoca el método, el viejoChar por el nuevoChar, todas las apariciones.
replaceAll(String viejaString, String nuevaString)	Reemplaza en el string que invoca al método la viejaString por la nuevaString. Se utiliza para reemplazar subcadenas.
replaceFirst (String viejaString, String nuevaString)	Reemplaza solo la primera aparición.

Colecciones

Conjunto de elementos que están almacenados de forma conjunta en una misma estructura de datos. La única restricción que tienen las colecciones es que los objetos que almacenan en su interior han de ser del mismo tipo, o de tipos relacionados.

En Java, para poder utilizar las colecciones necesitamos la clase `Collection` y el siguiente import:

```
import java.util.Collection;
```

Métodos genéricos

Método	Descripción
<code>int size()</code>	Devuelve la cantidad de elementos que tiene la colección.
<code>void add(Object ob)</code>	Agrega a la colección el elemento que se pasa por parámetro.
<code>void addAll(Collection c)</code>	Agrega a la colección todos los elementos de la colección que se pasa por parámetro.
<code>boolean remove(Object ob)</code>	Elimina de la colección el elemento que se pasa por parámetro.
<code>boolean removeAll(Collection c)</code>	Elimina todos los elementos de la colección que se encuentran en la colección que se pasa por parámetro.
<code>boolean isEmpty()</code>	Indica si la colección está vacía o no.
<code>void clear()</code>	Elimina todos los elementos de la colección.

Listas

Vamos a tender a usar la lista *ArrayList*, por su sencillez.

Este tipo de colección admite elementos repetidos en su interior, y añade las siguientes funcionalidades:

- **Acceso a los elementos por posición:** Se permite acceder a un elemento en concreto indicando su posición, , como en

un array.

- **Búsqueda de elementos:** Las listas permiten buscar elementos en su interior.
- **Operaciones con rangos:** Con las listas también se posibilita la opción de realizar operaciones indicando un rango de tamaño.
- **Iteración sobre los elementos:** Se permite iterar (recorrer los elementos mientras se hacen operaciones con ellos) sobre los elementos de una forma mucho más eficiente.

```
public static void main(String[] args) {  
    LinkedList<Integer> lista = new LinkedList<>();  
  
    for (int i = 0; i < 10; i++) {  
        lista.add(i);  
    }  
  
    boolean estaEl14 = lista.contains(14);  
    lista.remove(4);  
  
    System.out.println(lista);  
}
```

Métodos más comunes ARRAYLIST

Método	Descripción
int size()	Devuelve el número de elementos (int)
add(Object ob)	Añade el objeto X al final. Devuelve true.
add(int pos, Object ob)	Inserta el objeto X en la posición indicada.
Object get(int pos)	Devuelve el elemento que está en la posición indicada.
Object remove(int pos)	Elimina el elemento que se encuentra en la posición indicada. Devuelve el elemento eliminado.
Boolean remove(Object ob)	Elimina la primera ocurrencia del objeto X. Devuelve true si el elemento está en la lista.
clear()	Elimina todos los elementos.
set(int pos, Object ob)	Sustituye el elemento que se encuentra en la posición indicada por el objeto X.
Boolean contains(Object ob)	Comprueba si la colección contiene al objeto X. Devuelve true o false.
int indexOf(Object ob)	Devuelve la posición del objeto X. Si no existe devuelve -1.
boolean isEmpty()	Devuelve True si el <i>ArrayList</i> está vacío. Si no Devuelve False.
int lastIndexOf(Object ob)	Devuelve la última posición del objeto X. Si no existe devuelve -1.

Conjuntos (Set)

Esta clase no tiene ninguna funcionalidad nueva, teniendo que utilizar el siguiente import para poder utilizarla:

```
import java.util.set;
```

Para que esta clase funcione correctamente, es que los elementos con los que se vaya a utilizar deberán tener de forma obligatoria implementados los métodos equals y hashCode. Es decir, si creamos un Set de Persona, la clase Persona deberá tener esos dos métodos implementados.

Tipos deferentes de conjuntos:

- **HashSet:** Este tipo de conjunto utiliza tablas hash para su implementación, por lo que será bastante eficiente a la hora a de trabajar, aunque hay que destacar que no va a guardar los elementos siguiendo algún tipo de orden. Un inconveniente de este tipo de conjunto es que para que funcione correctamente necesita mucha memoria reservada.
- **TreeSet:** Este tipo de conjunto almacena los elementos ordenándolos por su valor, lo cual, hará que sea bastante más lento que el HashSet. Los elementos con los que se vaya a utilizar deberán implementar la interfaz Comparable.
- **LinkedHashSet:** Este tipo de conjunto utiliza para almacenar los elementos tablas hash y listas enlazadas (LinkedList). Haciendo uso de las tablas hash se conseguirá que sea rápido; con las listas enlazadas se conseguirá que los elementos se almacenen según el orden de inserción.

```
public static void main(String[] args) {  
    Set hashset = new HashSet();  
    for (int i = 0; i < 100000; i++) {  
        hashset.add(i);  
    }  
  
    System.out.println(hashset);  
    Set treeset = new TreeSet();  
    for (int i = 0; i < 100000; i++) {  
        treeset.add(i);  
    }  
  
    System.out.println(treeset);  
    Set linkedhashset = new LinkedHashSet(1_000_000);  
    for (int i = 0; i < 100000; i++) {  
        linkedhashset.add(i);  
    }  
    System.out.println(linkedhashset);  
}
```

Mapas

Los mapas, o *map* en inglés, son un tipo de colección que admite dos valores por elemento y que no puede contener elementos repetidos

Los mapas se basan en elementos de tipo clave-valor, esto quiere decir, que cada elemento que contenga el mapa tendrá tanto una clave como un valor, siendo el valor de la clave el identificador del valor.

Cada clave solo puede tener un elemento asociado, aunque si utilizamos un *ArrayList* (por ejemplo) en el campo de valor podremos tener más de un valor, respetando que una clave solo puede tener un valor.

Tipos de mapas:

- **HashMap:** Este tipo de mapa almacena sus elementos en una tabla hash, con lo que será muy rápida a la hora de realizar las operaciones. No hay ningún tipo de orden a la hora de almacenar los elementos en los HashMap. En el momento de crearlo habrá que definir su tamaño. No aceptan claves duplicadas ni valores nulos.
- **TreeMap:** Este tipo de mapa almacena los elementos ordenándolos por su valor, lo cual hará que sea bastante más lento que el HashMap. Los elementos con los que se vaya a utilizar deberán implementar la interfaz Comparable.
- **LinkedTreeMap:** En este caso, se utiliza para almacenar los elementos tablas hash y listas enlazadas, teniendo los mismos efectos que para los conjuntos, es decir, que haciendo uso de las tablas hash se conseguirá que sea rápido y con las listas enlazadas se conseguirá que los elementos se almacenen según el orden de inserción. Esta clase realiza las búsquedas de los elementos de forma más lenta que las demás clases.

A la hora de crear un mapa habrá que indicarle el tipo de elemento que tendrá tanto la clave como el valor.

```
public static void main(String[] args) {  
    Map<Integer, String> hashmap = new HashMap<>();  
    hashmap.put(1, "Valor uno");  
    System.out.println(hashmap);  
  
    Map<Integer, String> treemap = new TreeMap<>();  
    treemap.put(1, "Valor uno");  
    System.out.println(treemap);  
  
    Map<Integer, String> linkedhashmap = new LinkedHashMap<>();  
    linkedhashmap.put(1, "Valor uno");  
    System.out.println(linkedhashmap);  
}
```

Iteradores

Nos van a permitir acceder a los elementos de las colecciones, ya sean ArrayList, Set, Map...

El uso de iteradores para recorrer los elementos es obligatorio en mapas y conjuntos, mientras que en ArrayList no, pero son recomendables. Esto se debe a que no tienen orden, por lo que no se pueden recorrer con un for, como se hace con los arrays. Solo ArrayList dispone de un método para acceder a los elementos por su índice.

```
Iterator<tipo> nombrevariable = colección.iterator();
```

Nos van a proporcionar tres métodos:

- **next():** Este método devuelve el valor del elemento en el que está el iterador.
- **hasNext():** Este método indica si hay un elemento siguiente en el orden de iteración.
- **remove():** Este método eliminará un el elemento que tenga el iterador de la colección.

Los iteradores se pueden usar de forma transparente al usuario mediante el bucle *for-each*. Este tipo de bucle recorre una colección utilizando iteradores, pero sin que el programador tenga que definirlos. Su sintaxis es la siguiente:

```
for( tipo nombrevariable : Colecction )
```

- Tipo es el tipo de dato que contiene la colección.
- *Nombrevariable* es el nombre que le daremos a la variable iteradora.
- *Colecction* es la colección que vamos a recorrer.