**CHULETARIO PROGRAMACIÓN.**

**\*\* Pedir valor por teclado: \*\***

Scanner sc = new Scanner (System.in);

x = sc.nextInt();

**\*\* Estructura if: \*\***

if (condición) {

condición en true

} else {

condición en false

}

**\*\* Estructura condicional: \*\***

variable = (condición) ? condición true : condición false;

System.out.print((condición) ? condición true : condición false));

**\*\* Estructura switch: \*\***

switch (variable) {

case valorVariable: acción en ese valor

break;

default: Acción en case no recogida

}

**\*\* Estructura while: \*\***

while (condición en true) {

//código

} // hasta que la condición no se cumpla

**\*\* Estructura do-while: \*\***

do {

//código

} while (condición); //hacer minimo 1 vez, hasta que la condición se cumpla

**\*\* Estructura for: \*\***

for(definición contador(i=0) ; condición rango(i/</>/=/x) ; incremento contador(i++)){

//código

}

**\*\* MATEMÁTICAS: \*\***

|  |  |  |
| --- | --- | --- |
| Math.random() | Número aleatorio<<  Math.random()\*((mayor+1)-menor)+menor | números entre el 10 y el 50  x=50 y=10  Math. random ()\*(51-10)+10 |
| Math.pow(x,y) | Potencia  Math.pow(base,elevado) | Math.pow(10,2) = 100 |
| (int)Math.floor(x) | truncado  Math.floor(double) | x=2,25  Math.floor(x) = 2 |
| (int)Math.log10 | A qué número hay que elevar 10 para que de el número dado | x=123456 longitud de x = 6  (int)Math.log10(x) = 5 |

**\*\* Saber si un número es primo: \*\***

for (int i = 2; i < num; i++) {

if (num%i==0){

primo = false;

}

}

primo = true;

**\*\* Dar vuelta a un número: \*\***

int invertido=0, resto;

while (num > 0) {

resto = num%10;

invertido = invertido\*10+resto;

num/=10;

}

return invertido; - ( esto para función, sino sout)

**\*\* Maximo y minimo: \*\***

Int cont=0, max=0, min=0;

Int c = (loquesea);

if (cont==0){

max = c;

min = c;

} else {

if (max<c){

max = c;

}

if (min>c){

min = c;

}

}

**CADENAS:**

cadena.length() - Longitud de la cadena

cadena.equals(cadena2) // cadena.equals(“palabras”) - Igualar cadenas

cadena.equalsIgnoreCase - Igualar cadenas sin importar mayúsc o minusc

cadena.charAt() == ‘carácter’ // cadena.charAt(posición\_carácter) - Devuelve carácter en pos

cadena.substring(long1,long2) - Devuelve subcadena desde posición n1 hasta n2-1

cadena.indexOf(“frase/caracter”) == longitud - Devuelve primera posición del carácter

cadena.lastIndexOf(“frase”) == longitud - Devuelve última posición del carácter

cadena.compareTo(cadena2) - Devuelve si las cadenas son iguales o no

**\*\* Coger carácter del medio: \*\***

2 variables string para la frase num = (int) Math.floor((frase.length()-1)/2)

1 variable char para el carácter caracter = frase.charAt((int)num);

1 varible double para dividir en 2 la frase frase2 = frase2+caracter;

Para coger el medio If (frase2.equals(“ “))

**\*\* Contar espacios: \*\***

1 variable string para la frase

1 variable int para el contador

Recorremos la frase, if (frase.charAt(i)==’ ‘) -> contador++

**\*\* Eliminar caracteres: \*\***

2 variables string para la frase y el resultado

Recorremos la frase, if (frase.charAt(i)!=’carácter’ -> resultado += frase.charAt(i)

**\*\* Contar dígitos: \*\***

String cont = Integer.toString(num);

return cont.length();

**ARRAYS:**

Int[] variable = new int[num]; // Int[][] variable = new int[num][num];

Arrays.toString(variable) -> Mostrar vector

Array.sort(variable) -> Ordenar

Array.copyOf(variable, variable.length-1) -> copiar

Arrays.deepToString(variable) -> mostrar tabla bidimensional

**\*\* Unir vectores: \*\***

Creamos dos vectores normales, y el tercero con la suma de sus longitudes

int[] union = new int[num1.length+num2.length];

Bucle for para recorrer

Igualamos el vector unión al primer vector que queremos juntar

Igualamos el vector unión i + primer vector .length al segundo vector que queremos introducir

for (int i = 0; i < num1.length; i++) {

union[i] = num1[i];

}

for (int i = 0; i < num1.length; i++) {

union[i+num1.length] = num2[i]; }

**\*\* Sacar máximo y mínimo: \*\***

if (i==0){

max = numeros[i];

min = numeros[i];

} else {

if (max<numeros[i]){

max = numeros[i];

}

if (min>numeros[i]){

min = numeros[i];

}

}

**\*\* Insertar en la posición que quiera: \*\***

Creamos el array con los números

Creamos la función a la que insertamos el array v, int pos, int num

Creamos otro array auxiliar para insertar el num en la pos que quiera

public static void main(String[] args) {

int[] a = {3, 4, 5, 6, 7, 8};

System.out.println(Arrays.toString(insertar(a, 2, 9)));

}

public static int[] insertar(int[] v, int pos, int num) {

int[] aux = new int[v.length + 1];

for (int i = 0; i < aux.length; i++) {

if (i < pos) {

aux [i] = v[i]; // antes de la pos insertamos los mismos números que ya estaban

} else if (i == pos) {

aux [i] = num; // si es igual, insertamos el número

} else {

aux [i] = v[i - 1]; // insertamos el resto de números

}

}

return aux;

}

**\*\* Insertar en la primera posición y rotar: \*\***

Para la primera posición aux[0] = num;

Recorremos el vector con un for de v.length dónde aux[i+1] = v[i] porque la primera posición ya la hemos rellenado con lo anterior

Para la última posición recorremos con un for de v.length dónde aux[i] = v[i] y fuera del for ponemos la posición aux[v.length] = num;

**\*\*CON ARRAYS.COPYOF\*\***

**\*insertar número al principio y rotar los demás\*\***

aux = new type[0];

aux = Arrays.copyOf(aux, aux.length+1);

aux[aux.length-1]=num;

for(int i=0; i<v.length; i++){

aux = Arrays.copyOf(aux, aux.length+1);

aux[aux.length-1]=v[i];

}

**\*\*asignar los valores de aux a v\*\***

v = aux;

**\*\* Borrar elemento: \*\***

Igual que insertar, pero cuando i>pos restamos una posición para eliminar la que hayamos elegido aux[i-1]

for (int i = 0; i < aux.length; i++) {

if (i < pos) {

aux [i] = v[i];

} else if (i > pos) {

aux [i-1] = v[i];

}

}

**\*\* Con Arrays.copyOf: \*\***

Usamos un parámetro para hacer la comparación con el if, por ej num. Si el num que introduzco no es igual a los demás, hago el array copy of y luego lo igualo a un nuevo array aux.

Int[] aux = new aux[0];

for (int i = 0; i < array.length; i++) {

if (num!=array[num2]) {

aux = Arrays.copyOf(aux, aux.length+1);

aux [aux.length-1] = array [i];

}

array = aux;

}

**\*\* Mostrar en orden inverso: \*\***

for (int i = num.length-1; i >= 0 ; i--) {

System.out.print(num [i]);

}

**\*\* Pares al principio, impares al final: \*\***

public static void paresimpares(int[] v){

int[] par = new int[0];

int[] impar = new int[0];

for (int i = 0; i < v.length; i++) {

if (v[i]%2==0){

par = Arrays.copyOf(par,par.length+1);

par[par.length-1] = v[i];

} else {

impar = Arrays.copyOf(impar,impar.length+1);

impar[impar.length-1] = v[i];

}

}

for (int i = 0; i < v.length; i++) {

if (i<par.length){

v[i] = par[i];

} else {

v[i] = impar[i-par.length];

}

}

int[] union = new int[pares.length+impares.length];

for (int i = 0; i < pares.length; i++) {

union[i] = pares[i];

}

for (int i = 0; i < impares.length; i++) {

union[i+pares.length] = impares[i];

}

System.out.println(Arrays.toString(union));

}

**\*\* Pasar números a cadena: \*\***

variable String vacía

bucle for para recorrer vector

variable String += vector

return String

**\*\* Recorrer vector bidimensional y darle valores: \*\***

* Si son num aleatorios:

for (int i = 0; i < vector.length; i++) {

for (int j = 0; j < vector[i].length; j++) {

vector[i][j] = (int)(Math.random()\*101);

}

}

* Si son datos pedidos por teclado:

Int(u otro tipo)[][] vector;

sout para pedir el tamaño del vector

vector = new int[filas][columnas];

Bucle for pidiendo datos

**\*\* Suma de fila/columna/diagonal de matriz: \*\***

int suma = 0;

int fila = 0;

int columna = 0;

// Fila

for (int i = 0; i < b1.length; i++) {

for (int j = 0; j < b1[i].length; j++) {

suma += b1[fila][j];

}

}

// Columna

for (int i = 0; i < b1.length; i++) {

for (int j = 0; j < b1[i].length; j++) {

suma += b1[i][columna];

}

}

// Diagonal

for (int i = 0; i < b1.length; i++) {

for (int j = 0; j < b1[i].length; j++) {

if (i==j){

suma += b1[i][j];

}

}

}

**\*\* Rotar posiciones: \*\***

public static void rotar(int[][] n) {

int[] resultado = new int[0];

for (int i = 0; i < n.length; i++) {

for (int j = 0; j < n[i].length; j++) {

resultado = Arrays.copyOf(resultado,resultado.length+1);

resultado[resultado.length-1] = n[i][j]; // Para aumentar tamaño matriz

}

}

int cont1=0,cont2=0;

for (int i = 0; i < resultado.length; i++) { // Rotar unidimensional

if (i==0) {

cont1 = resultado[i]; // cont1 guarda el primer valor que tenemos

resultado[i]=resultado[resultado.length-1]; // guardamos el último valor

} else {

cont2 = resultado[i]; // guarda el valor de la posición i

resultado[i] = cont1; // meter el valor de la posición i-1

cont1=cont2; // guardar el valor siguiente

}

}

int cont3=0;

for (int i = 0; i < n.length; i++) { // Para meter valor rotado

for (int j = 0; j < n[i].length; j++) {

n[i][j] = resultado[cont3]; // meter valor de unidimensional a bidimensional

cont3++; // contador para ir cogiendo todos los valores del unid

}

}

}

**\*\* Rellenar con números aleatorios sin que se repitan: \*\***

int[][] cubo = new int[5][5];

int num;

for (int i = 0; i < cubo.length; i++) {

for (int j = 0; j < cubo[i].length; j++) {

do {

num = (int) (Math.random()\*40);

if (!siEsta(cubo, i, j, num)) {

cubo[i][j] = num;

}

} while (cubo[i][j]==0);

}

}

for (int i = 0; i < cubo.length; i++) {

System.out.println(Arrays.toString(cubo[i]));

}

}

public static boolean siEsta(int[][] array, int altura, int anchura, int num) {

for (int i = 0; i < altura; i++) { // bucle filas completas

for (int j = 0; j < array[0].length; j++) {

if (num == array[i][j]) {

return true;

}

}

}

for (int i = 0; i < anchura; i++) { // bucle filas incompletas

if (num==array[altura][i]){

return true;

}

}

return false;

}

**\*\* Buscar numero dentro de un array: \*\***

public static int nEsimo(int[][] n, int posicion) {

int numero = 0;

for (int i = 0; i < n.length; i++) {

for (int j = 0; j < n[i].length; j++) {

if (numero == posicion) {

return n[i][j];

}

numero++;

}

}

return -1;

}

**PROGRAMACIÓN ORIENTADA A OBJETOS:**

**\*\* Clases, objetos, atributos y métodos: \*\***

Declaramos las clases que vamos a usar en el main, con sus respectivos atributos inicializados:

public class nombre {

tipo atributo1 = valor;

tipo atributo2 = valor;

}

Si no inicializamos los atributos, podemos darle valor en el main, cuando creamos los objetos:

(nombre\_clase) (variable) = new (nombre\_clase)();

variable.atributo1 = “ “;

variable.atributo2 = 0;

Podemos asignar métodos dentro de las clases igual que atributos:

tipo nombre\_metodo (parámetros) {

códigos con que realizará el método

}

\*\* Ejemplo: \*\*

public class Persona {

String nombre;

byte edad;

double estatura;

void saludar() {

System.out.println(“Hola, mi nombre es “+nombre+” ,encantado.”);

}

void cumplirAños() {

edad++;

}

void crecer(double incremento) {

estatura += incremento;

}

}

Persona p = new Persona();

p.edad = 18;

p.cumplirAños();

**\*\* Ocultación de atributos: \*\***

Cuando las variables tienen el mismo nombre en el método y en el atributo, tiene prioridad la del método, por lo que si hacemos una llamada a esa variable se mostrará la local (método) y se oculta la del atributo.

**\*\* Objeto this: \*\***

Permite usar un atributo incluso cuando ha sido ocultado.

this.variable = “ “ ;

Permite llamar a otro constructor

this(atributo1, atributo2,...);

**\*\* Atributos y métodos estáticos: \*\***

Atributo estático = no existe una copia en cada objeto, pero se debe compartir entre todos los objetos, ya que el valor será el mismo para todos.

* Por ej: debemos poner el día de la semana que es hoy en la clase Persona

class Persona {

…

static String hoy = “lunes”;

}

También podemos poner static String hoy; y darle el valor al crear el objeto (Persona.hoy=”lunes”).

Método estático = no requieren ningún objeto para ejecutarse y no pueden usar atributos que no sean estáticos

* Por ej: método que actualice el atributo hoy a partir de un entero (1-7) que representa los días

static void hoyEs(int dia) {

switch (dia) {

case 1:

hoy = “lunes”;

break;

case 2:

hoy = “martes”;

break;

………

case 7:

hoy = “domingo”;

break;

}

}

Persona.hoyEs(2); //martes

Si tenemos un objeto Persona creado con la variable p, podemos poner [ p.hoyEs(2); ] que hace lo mismo que la línea anterior, actualizar el día.

**\*\* Constructores: \*\***

Constructor = método especial que debe tener el mismo nombre que la clase, se define sin tipo devuelto y se ejecuta después de crear el objeto. Sirven para asignar valores a los atributos, aunque también se puede utilizar para ejecutar cualquier código necesario.

* Por ej: Vamos a implementar un constructor para Persona que asigne los valores iniciales de sus atributos. (fn+alt+ins impr pa para crear en intelliJ)

public class Persona {

…

Persona (String nombre, int edad, double estatura) {

this.nombre = nombre;

this.edad = edad;

this.estatura = estatura;

}

…

}

Los valores se asignan al crear el objeto con new.

Persona p = new Persona (“Claudia”, 8, 1.20);

Si solo conocemos el valor de una variable, solo se usa el constructor con esa variable.

Cuando una clase dispone de varios constructores se pueden invocar unos a otros.

* Por ej: un constructor que asigan todos los valores a los atributos y otro que solo al nombre

public class Persona {

…

Persona (String nombre, int edad, doublé estatura) {

this.nombre = nombre;

this.edad = edad;

this.estatura = estatura;

}

Persona (String nombre) {

this(nombre, 18, 1.65);

}

}

**\*\* Modificadores de acceso: \*\***

Una clase será visible por otra dependiendo si se ubican en el mismo paquete y de los modificadores de acceso que utilice. Es posible modificar la visibilidad de atributos y métodos para otras clases.

* Modificadores de acceso para clases:
  + Clases vecinas: cuando están en el mismo paquete
  + Clases externas: cuando estén en paquetes distintos

package paquete1; import paquete1.A; import paquete1.\*;

public class A { class C {

… …

} }

* Modificadores de acceso para miembros:

Para que un miembro sea visible, su clase también debe serlo. Un miembro es siempre visible dentro de su propia clase (dentro de una clase podremos invocar cualquiera de sus métodos).

* Visibilidad por defecto:

Un miembro es visible desde las clases vecinas (accede a clase y sus atributos), pero invisible desde clases externas (accede a la clase pero no a sus atributos). Si la clase se define como public permite que sea visible desde clases externas.

* Modificador de acceso private y public: con private se invoca desde otros métodos de la clase, pero no fuera de ella, con public es visible desde clases externas también.

**\*\* Getters y Setters: \*\***

getNombreAtributo(); devuelve el valor del atributo sin exponerlo.

setNombreAtributo(Type atributo){

this.nombreAtributo = atributo;

}

**\*\* Enum: \*\***

Son listas cerradas, con las que no se puede elegir otros valores que no esten en el enum

Ejemplo:

public enum Colores{

ROJO, VERDE, AZUL, AMARILLO, GRIS

}

**\*\* Static: \*\***

Un método estático es una función que no depende de los atributos del objeto. puede recibir parámetros.

**\*\* Herencias: \*\***

La herencia es la modulación de de atributos, donde los padres comparten sus atributos y métodos con los hijos y todos sus descendientes.

**\*\* Extends:\*\***

Para asignar un padre a una clase, usamos la variable reservada -- ***extends clasePadre*** --

Para compartir los atributos sin poner en riesgo sus datos, en la clase padre cambiamos la visibilidad de los atributos a ***protected***.

todos los atributos y métodos del padre pueden ser llamados por el hijo.

\*\***override:\*\***

Modifica el cuerpo de un método ya implementado en el padre, que en el hijo realiza unas funciones diferentes a la ya creada.

@Override

public class mismoNombredeClasePadre(type MismosParametros, …)

**\*\* Abstract:\*\***

Son clases que no se pueden instanciar(Hacer **new**) deben ser clases padres, y ayudan a insertar todos los hijos en un array de **type clasePadre.**

Se pueden crear cabeceras que tendrán todos los hijos, pero no se rellena en las clases padre o hijas abstractas

ejemplo

public abstract class NombreClasePadre{

…

public abstract void llamar(String telefono);

}

public class NombreClaseHija{

…

public abstract void llamar(String telefono){

//codigo

}

**\*\* Interfaces-implement methods: \*\***

Las interfaces funcionan como las clases abstractas, pero con la diferencia de que puedes añadir todas las interfaces que quieras a una clase, y no depende del padre para poder implementarla.

Todas las funciones de las interfaces deben ser implementadas en la clase en la que se le llame.

se añaden a nuestra clase usando en la cabecera de la clase -- implements nombreInterfaces

si se quiere añadir varias interfaces, se añaden las interfaces separadas por comas.

-- implements nombreInterfaces, nombreInterfaces2, ...

**\*\* Comparar: \*\***

comparable = implements Comparable<Objeto> / compareTo this.atrib-objeto.abtrib / ordenar con Arrays.sort(); ascendente, para cambiar a descendente objeto.abtrib-this.atrib

**// Comparacion por defecto, con metodo en la clase creada**

Arrays.sort(clientes);

System.out.println(Arrays.toString(clientes));

**// Comparacion aparte en una nueva clase (implements Comparator<Clase>)**

Arrays.sort(clientes,new ComparadorEdades());

System.out.println(Arrays.toString(clientes));

**// Comparación directamente llamando al método**

Arrays.sort(clientes, new Comparator<Cliente>() {

@Override

public int compare(Cliente o1, Cliente o2) {

return o1.getNombre().compareToIgnoreCase(o2.getNombre());

}

});

**\*\* instanceOf: \*\***

Para coger un objeto hijo de una clase padre. Ej:

public int compare(Vehiculos o1, Vehiculos o2) {

if (o1 instanceof Terrrestre){

if (o2 instanceof Terrrestre){

return ((Terrrestre) o1).matricula.compareToIgnoreCase(((Terrrestre) o2).matricula);

} else {

return ((Terrrestre) o1).matricula.compareToIgnoreCase(((Aereo) o2).oaci);

}

}

**\*\* getClass() y getSimpleName(): \*\***

La primera te devuelve toda la arquitectura del objeto desde padre hasta hijos.

La segunda te devuelve el nombre de la clase hijo.

Se usan las dos juntas.

\*\* Cast: \*\*

**\*\* Excepciones: \*\***

try {

// CODIGO QUE PUEDE GENERAR EL ERROR

} catch (Exception e){

// REACCION AL ERROR

}

**\*\* Excepciones con ficheros: \*\***

try {

// CODIGO QUE PUEDE GENERAR EL ERROR

return

} catch (IOException e){

// REACCION AL ERROR

} finally {

// CODIGO PARA CERRAR ARCHIVOS

}

Return;

**\*\* FileReader: \*\***

FileReader in = new FileReader(“fichero”);

int c = in.read(); - pasa a la siguiente posición / lee de 1 en 1

in.close(); - cuando acabo de leer

String contenido = “ ”;

try {

FileReader in = new FileReader(“./ruta/fichero.txt”);

int c = in.read();

while (c!=-1){

contenido += (int) c;

c = in.read();

}

In.close();

} catch (IOException e){

System.out.println(e.getMessage());

}

System.out.println(contenido);

**\*\* BufferedReader: \*\***

Para leer trocito a trocito

BufferedReader in = new BufferedReader(new FileReader(“./ruta/fichero.txt”));

try {

BufferedReader in = new BufferedReader(new FileReader(“fichero.txt”));

String contenido = “ “;

String linea = in.readLine();

while (linea!=null){

contenido += línea+”\n”;

línea = in.readLine();

}

In.close();

} catch (IOException e){

System.out.println(e.getMessage());

}

System.out.println(contenido);

**\*\* FileWriter y BufferedWriter: \*\***

try {

FileWriter out = new FileWriter("nuevoejemplo.txt", true); // true = machacar lo que ya tenias (escribir a partir de lo anterior, sino se borra)

BufferedWriter bw = new BufferedWriter(out);

bw.write(97);

bw.newLine();

bw.write("DIOS");

bw.flush(); // para guardar los cambios que hemos hecho anteriormente en el fichero

bw.close();

} catch(IOException e){

e.printStackTrace();

}

**\*\* Convertir: \*\***

Al usar Buffered reader y querer sacar el max y el minimo de una lista de numeros, pasamos línea que es String a Integer.

* int c = Integer.parseInt(linea);

**\*\* Comprobar si se encuentra: \*\***

Utilizando funciones como hemos anteriormente:

public static boolean estaFirmado(String firma){

* Dentro del try / catch, en el while:

while (linea!=null){

if (linea.equalsIgnoreCase(firma)){

return true;

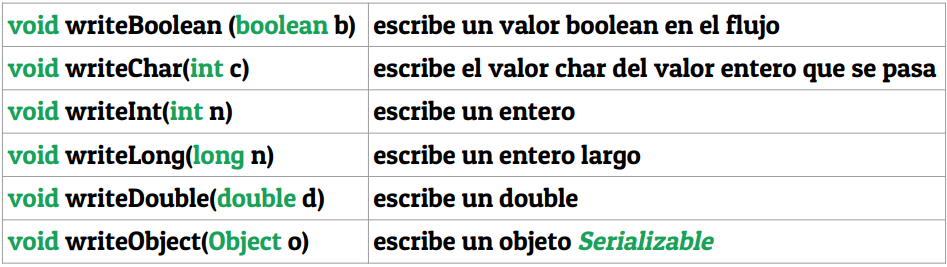
}

return false;

**\*\* Ficheros binarios: \*\***

**\*\* ESCRIBIR: \*\***

ObjectOutputStream - FileOutputStream



**Para escribir en un fichero binario, ejemplo:**

public class Binarios {

public static void main(String[] args) {

Persona p = new Persona("Pepa","Perez");

int[] num = {1,2,3,4,5,6,7,8};

try{

ObjectOutputStream flujoSalida = new ObjectOutputStream(new FileOutputStream("./src/U6\_T2/prueba.abg"));

flujoSalida.writeInt(50);

flujoSalida.writeObject(p);

flujoSalida.writeObject(num);

flujoSalida.close();

} catch (IOException e) {

e.printStackTrace();

}

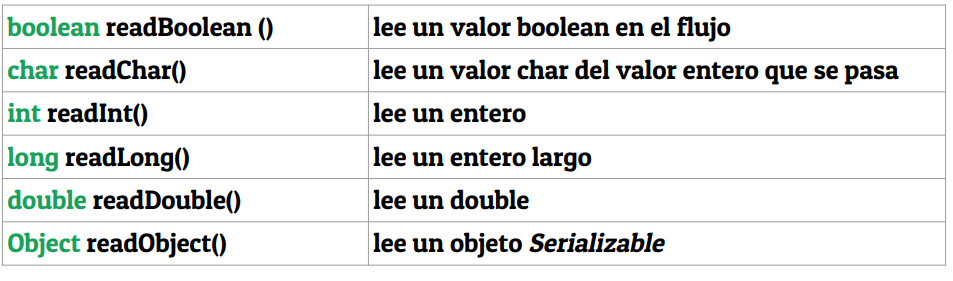
}

}

En el objeto Persona tenemos que poner ***implements Serializable***, sino daría error.

**\*\* LEER: \*\***

ObjectInputStream - FileInputStream



Usando el ejemplo anterior, hay que seguir el mismo orden para leer que lo que hemos puesto para escribir:

try{

ObjectInputStream flujoEntrada = new ObjectInputStream(new FileInputStream("./src/U6\_T2/prueba.abg"));

System.out.println(flujoEntrada.readInt());

System.out.println((Persona)flujoEntrada.readObject());

System.out.println(Arrays.toString((int[])flujoEntrada.readObject()));

} catch (IOException | ClassNotFoundException e){

e.printStackTrace();

}

**\*\* EOFException: \*\***

try {

ObjectInputStream flujoEntrada = new ObjectInputStream(new FileInputStream("./src/U6\_T2/prueba.abg"));

while (true) {

System.out.println(flujoEntrada.readInt());

}

} catch (EOFException | FileNotFoundException ex) {

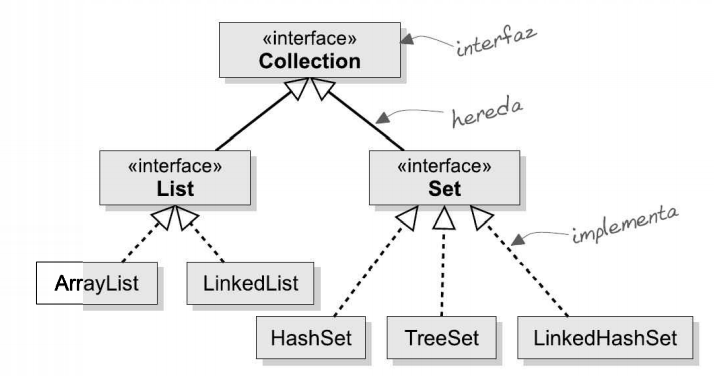
System.out.println("Fin de fichero");

} catch (IOException e) {

e.printStackTrace();

}

**\*\* COLLECTION: \*\***



**LIST** Guarda una serie de datos que pueden estar repetidos y el orden puede ser importante.

**SET** Guarda una serie de datos sin repetir y donde el orden no es importante.

**\*\* LIST: \*\***

ArrayList: ↑Leer

↑Modificar

LinkedList: ↑Insertar

↑Eliminar

*boolean add(Element e)* : Inserta “e” al final del ArrayList.

* True si se modifica el arrayList(se añade).
* False si no se modifica el arrayList(no se añade).

*void add(int index, Element e)* : Inserta en la posición index y desplaza los elementos desde esa posición a index+1

*boolean remove(Element e)* : Elimina El primer elemento que sea igual al pasado por parámetro.

* True si se modifica el arrayList(se elimina).
* False si no se modifica el arrayList(no se elimina).

*void clear()* : Elimina todos los elementos del arrayList.

*int size()* : Longitud del arrayList.

*boolean isEmpty()* : Si está o no vacío el arrayList.

* True si no tiene elementos(size()==0).
* False si tiene algún elemento(size()>0).

*boolean contains(Element e)* :

* True si se encuentra el elemento en el arrayList.
* False si no encuentra el elemento en el arrayList.

E get(int index) : Devuelve el elemento en la posición index.

*E set(int index, Element e)* : Devuelve el elemento en la posición index y lo cambia por el pasado por parámetro.

*int indexOf(Object o)* : Devuelve el índice de la primera vez que encuentra el objeto.

*int lastIndexOf(Object o)* : Devuelve el índice de la última posición del objeto.

*boolean equals(Collection<E> c)* : compara los valores en las mismas posiciones.

* True si todos los valores tienen los mismos datos en las mismas posiciones con la lista que se le pasa.
* False si algún elemento no es igual.

*boolean ConteinsAll(Collection<?> c)* : Si contiene o no todos los elementos del arrayList pasado por parámetro.

*boolean addAll(Collection<?> c)* : Añade al final todos los elementos del arrayList pasado.

*boolean RemoveAll(Collection<?> c)* : Elimina todos los elementos que coincidan con el arrayList pasado.

*boolean retainAll(Collection<?> c)* : Elimina todos los elementos que no coincidan con el arrayList pasado.

*Object[] toArray(Object[] o)* : Se le pasa por parámetro un array del mismo tipo con longitud 0.

**\*\* ITERATOR: \*\***

Iterator<E> it = new lista.iterator();

*boolean hasNext();* Comprueba si hay más registros en el arrayList

*Object next();* devuelve el siguiente registro del arrayList

for( Iterator it = lista.iterator(); it.hasNext(); ){

Object o = (o) it.next();

}

**\*\* SET: \*\***

HashSet:↑Rendimiento Sin orden en la inserción.

TreeSet: ↓Rendimiento Orden basado en el compareTo del objeto o por el compare del comparator.

LinkedHashSet: Garantiza el orden por inserción. Siempre se inserta al final.

**\*\* Conversión Collection: \*\***

*addAll(Collection)*

Set<Integer> s1 = new LinkedHashSet<>();

s1.add(..);

s1.add(..);

s1.add(..);

s1.add(..);

Set<Integer> s2 = new TreeSet<>();

s2.addAll(s1);

*Constructor(Collection)*

Set<Integer> s1 = new LinkedHashSet<>();

s1.add(..);

s1.add(..);

s1.add(..);

s1.add(..);

Set<Integer> s2 = new TreeSet<>(s1);

**\*\* Collections: \*\***

Son métodos estáticos para hacer Ordenación, Búsqueda, etc.

List<Integer> lista = new ArrayList<>();

Para ordenar por orden natural: *Collections.sort(list);*

Para buscar: Tiene que estar ORDENADO, Solo sirve para List(Set no tiene índices):

*Collections.binarySearch(list,CODIGO\_ORDENACIÓN);*

Intercambiar datos en dos posiciones:

[0,1,2,3,4]

*Collections.swap(0,3);*

[3,1,2,0,4]

Reemplazar un valor por otro:

*Collections.replaceAll(list,1,100);* → (lista, antiguoValor, NuevoValor)

Rellena una lista:

*Collections.fill(list, 500)* → (lista, valor)

Cantidad de repeticiones en una lista:

*Collections.frecuency(list,500);*