¿QUÉ ES UN PROGRAMA JAVA?

- Un programa Java es una colección de clases. Algunas clases las realizamos nosotros y otras forman parte del lenguaje Java.
- Un programa Java debe contener un método estático denominado **main** ( )
- Para trabajar con Java necesitamos **JDK** (Java Development Kit)
- Herramientas del **JDK**:

Javac	Compilador Java
Java	Interprete Java, utilizado para ejecutar programas compilados
Aplletviewer	Utilizado para visualizar el applet tal como puede ser visto por el navegador
JDb	Depurador
Javadoc	Generador de documentación

PALABRAS RESERVADAS

- Ciertas palabras están reservadas para uso interno por Java y no se pueden utilizar como nombres de variables.

abstract	**continue**	**for**	**new**	**switch**
assert	**default**	**goto**	**package**	**synchronized**
boolean	**do**	**if**	**private**	**this**
break	**double**	**implements**	**protected**	**throw**
byte	**else**	**import**	**public**	**throws**
case	**enum**	**instanceof**	**return**	**transient**
catch	**extends**	**int**	**short**	**try**
char	**final**	**interface**	**static**	**void**
class	**finally**	**long**	**strictfp**	**volatile**
const	**float**	**native**	**super**	**while**

- Además de estas palabras reservadas, Java se reserva **false,null y true** como valores definidos en el lenguaje

IDENTIFICADORES

- Un identificador es el nombre de variables, métodos, clases e interfaces.
- Un identificador es una secuencia ilimitada de caracteres alfabéticos o dígitos (unicode) que comienzan con un carácter alfabético

RESUMEN		
Elemento	**Nomenclatura**	**Ejemplo**
Paquete	minúscula	metodos, vistas, controladores
Nombre clase	1ª letra mayúsculas+minúscula+1ª letra de cada palabra: mayúsculas	DatosEmpleado,
Métodos y variables	1ª letra minúsculas+ 1ª letra de cada palabra: mayúsculas	introducirDatosEmpleado totalArticulos, setNombre
Constantes	Todo en mayúsculas	PI, NUMERO_HORAS

DECLARACIÓN DE CONSTANTES Y ENUMERADOS

- **Constante**: la declaración contiene la palabra final delante:
  final <tipo_datos> <nombre_constante> = <valor>;

- Ejemplo:
  ```
  final int MAX_ELEM = 20;
  ```
- **Tipo enumerado:** Java crea una nueva clase (puede tener campos y métodos):
  enum <nombre_enumeracion> {<nombre_constante>, ..., <nombre_constante> }

DECLARACIÓN DE VARIABLES Y TIPOS DE DATOS

- Una **variable** es una zona en la memoria del computador con un valor que puede ser almacenado para ser usado más tarde en el programa.
- Las variables vienen determinadas por:
  - Un **nombre**, que permite al programa acceder al valor que contiene en memoria. Debe ser un **identificador** válido.
  - Un **tipo de dato**, que especifica qué clase de información guarda la variable en esa zona de memoria
  - Un rango de valores que puede admitir dicha variable.
- Formato para declarar una variable:

  <tipo_datos> <nombre_variable>;

- Distingue **mayúsculas de minúsculas**

TIPOS DE DATOS PRIMITIVOS

<tipo_datos_primitivo> <nombre_variable> [= <valor>];

Tipo	Tamaño	Rango
byte	8 bits	−128 ... 127
short	16 bits	-32.768 ... 32.767
int	32 bits	-2.147.483.648 ... 2.147.483.647
long	64 bits	-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807
float	32 bits	-3.4×10^{38} ... 3.4×10^{38} (mínimo 1.4×10^{-45})
double	64 bits	-1.8×10^{308} ... 1.8×10^{308} (mínimo 4.9×10^{-324})
boolean		true o false
char	16 bits	Unicode

TIPO DE DATOS CADENA DE CARACTERES (STRING)

String <nombre_variable> [= "<cadena de caracteres>"];

TIPOS DE DATOS ENVOLVENTES

- Existe una **clase** equivalente para cada tipo de dato primitivo

Tipo Simple	Clase Equivalente
byte, short, int, long	Number, Byte, Short, Integer, Long
float, double	Number, Float, Double
boolean	Boolean
Char	Character

OPERADORES

OPERADOR DE ASIGNACIÓN

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
=	Operador asignación	`n = 4`	`n vale 4`

OPERADOR DE ASIGNACIÓN COMBINADOS

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
+=	Suma combinada	`a+=b`	`a=a+b`
-=	Resta combinada	`a-=b`	`a=a-b`
*=	Producto combinado	`a*=b`	`a=a*b`
/=	División combinada	`a/=b`	`a=a/b`
%=	Resto combinado	`a%=b`	`a=a%b`

OPERADORES ARITMÉTICOS

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
-	operador unario de cambio de signo	`-4`	`-4`
+	Suma	`2.5 + 7.1`	`9.6`
-	Resta	`235.6 - 103.5`	`132.1`
*	Producto	`1.2 * 1.1`	`1.32`
/	División (tanto entera como real)	`0.050 / 0.2` `7 / 2`	`0.25` `3`
%	Resto de la división entera	`20 % 7`	`6`

OPERADORES ARITMÉTICOS INCREMENTALES

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo

++	Incremento
	i++ primero se utiliza la variable y luego se incrementa su valor
	++i primero se incrementa el valor de la variable y luego se utiliza
		4++
		a=5;
		b=a++;
		a=5;
		b=++a;
			5
			
			a vale 6 y b vale 5
			
			a vale 6 y b vale 6

--	decremento	4--	3

OPERADOR CONDICIONAL

- El operador condicional **"?:"** sirve para evaluar una condición y devolver un resultado en función de si es verdadera o falsa dicha condición.

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo

?:	operador condicional	a = 4;
		b = a == 4 ? a+5 : 6-a;
		b = a > 4 ? a*7 : a+8;
			b vale 9
			b vale 12

OPERADORES DE RELACIÓN

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
==	igual que	7 == 38	false
!=	distinto que	'a' != 'k'	true
<	menor que	'G' < 'B'	false
>	mayor que	'b' > 'a'	true
<=	menor o igual que	7.5 <= 7.38	false
>=	mayor o igual que	38 >= 7	true

OPERADORES LÓGICOS

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
!	Negación - NOT (unario)	!false !(5==5)	true false
|	Suma lógica – OR (binario)	true | false (5==5)|(5<4)	true true
^	Suma lógica exclusiva – XOR (binario)	true ^ false (5==5)|(5<4)	true true
&	Producto lógico – AND (binario)	true & false (5==5)&(5<4)	false false
||	Suma lógica con cortocircuito: si el primer operando es true entonces el segundo se salta y el resultado es true	true || false (5==5)||(5<4)	true true
&&	Producto lógico con cortocircuito: si el primer operando es false entonces el segundo se salta y el resultado es false	false && true (5==5)&&(5<4)	false false

CONVERSIÓN DE TIPOS- CASTING

- Imagina que queremos dividir un número entre otro. Si el denominador y el divisor son variables de tipo entero, el resultado será entero y no tendrá decimales. Para que el resultado tenga decimales necesitaremos hacer una conversión de tipo.
- Las conversiones de tipo se realizan para hacer que el resultado de una expresión sea del tipo que nosotros deseamos.

```
int a,b;
float d;
d=(float)a/b;
int a;
byte b;
a = 12; // no se realiza conversión alguna
b = 12; // se permite porque 12 está del rango permitido de valores para b
b = a; // error, no permitido (incluso aunque // 12 podría almacenarse en un byte)
byte b = (byte) a; // Correcto, forzamos conversión explícita
```

Tabla de Conversión de Tipos de Datos Primitivos

		Tipo destino							
		boolean	**char**	**byte**	**short**	**int**	**long**	**float**	**double**
Tipo origen	**boolean**	-	N	N	N	N	N	N	N
	char	N	-	C	C	Cl	Cl	Cl	Cl
	byte	N	C	-	Cl	Cl	Cl	Cl	Cl
	short	N	C	C	-	Cl	Cl	Cl	Cl
	int	N	C	C	C	-	Cl	Cl*	Cl
	long	N	C	C	C	C	-	Cl*	Cl*
	float	N	C	C	C	C	C	-	Cl
	double	N	C	C	C	C	C	C	-

ENTRADA POR TECLADO

- En *java* tenemos accesible el teclado desde ***System.in***,

Clase: SCanner

- La clase **Scanner** permite leer datos a través del teclado
- Hay que impotar: **import java.util.Scanner;**

Scanner teclado=new Scanner(System.ini);

Método	Significado	Ejemplo
nextByte()	Número entero	`Byte num=teclado.nextByte();`
nextInt()	Número entero	`int entero=teclado.nextInt();`
nextLong()	Número entero	`Long l=teclado.nextLong();`
nextDouble()	Número real	`double real2=teclado.nextDouble();`
nextFloat()	Número real	`float real1=teclado.nextFloat();`
next()	String	`String nombre=teclado.next()`
nextLine()	String (quitando \n)	`String nombre=teclado.nextLine()`

Clases: InputStreamReader y BufferedReader

- **InputStreamReader**: es capaz de convertir los *bytes* a caracteres. **BufferedReader**: es capaz de leer hasta un fin de línea.
- La forma de instanciar estas clases para usarlas con *System.in* es la siguiente:
```
InputStreamReader entrada = new InputStreamReader(System.in);
BufferedReader teclado = new BufferedReader (entrada);
```
- También se puede poner:
```
BufferedReader teclado = new BufferedReader (new InputStreamReader(System.in));
```
- Para leer una línea por teclado:
```
String cadena = teclado.readLine();
```
- Podemos leer una cadena y convertirlo a entero:
```
int numero=Integer.parseInt(teclado.readLine());
```

SALIDA POR PANTALLA

- En *java* tenemos accesible la pantalla desde *System.in*,

println y print

- Para mostrar texto por pantalla en Java no es necesario hacer uso de librerías importadas, pues éstas ya están incluidas en la librería java.lang que siempre es importada por defecto.
- Para imprimir por pantalla, se usa la clase ***System***, el atributo ***out***, y su método ***println()*** o ***print()*** así *System.out.println()* o *System.out.print()*
- **println**():muestra por pantalla la cadena que haya entre paréntesis y realiza un salto de línea.
- **print**():muestra por pantalla la cadena que haya entre paréntesis sin realizar un salto de línea.
- Ejemplos: `System.out.println("Esto es lo que sale por pantalla");`

printf

- **printf** (que deriva su nombre de "*print formatted*") imprime un mensaje por pantalla utilizando una "cadena de formato" que incluye las instrucciones para mezclar múltiples cadenas en la cadena final a mostrar por pantalla.

  printf("cadena de formato",parámetros);
- En la cadena de formato se incluye texto a imprimir literalmente y ***marcas*** a reemplazar por texto que se obtiene de los parámetros adicionales.
- El símbolo "%" en la cadena de formato denota el comienzo de la marca de formato.
- Las marcas en la cadena de formato deben tener la siguiente estructura (los campos entre corchetes son optativos):

  %[parameter][flags][width][.precision][length]type

- Toda marca, por tanto, comienza por el símbolo "%" y termina con su tipo. Cada uno de los nombres (*parameter, flags, width, precision, length* y *type*) representa un conjunto de valores posibles que se explican a continuación.

Parameter	Descripción
n$	Se reemplaza "n" por un número para cambiar el orden en el que se procesan los argumentos. Por ejemplo %3$d se refiere al tercer argumento independientemente del lugar que ocupa en la cadena de formato.
Flags	**Descripción**
número	Rellena con espacios (o con ceros, ver siguiente flag) a la izquierda hasta el valor del número.
0	Se rellena con ceros a la izquierda hasta el valor dado por el flag anterior. Por ejemplo "%03d" imprime un número justificado con ceros hasta tres dígitos.
+	Imprimir el signo de un número
-	Justifica el campo a la izquierda (por defecto ya hemos dicho que se justifica a la derecha)
#	Formato alternativo. Para reales se dejan ceros al final y se imprime siempre la coma. Para números que no están en base 10, se añade un prefijo denotando la base.
Width	**Descripción**
número	Tamaño del ancho del campo donde se imprimirá el valor.
*	Igual que el caso anterior, pero el número a utilizar se pasa como parámetro justo antes del valor. Por ejemplo printf("%*d", 5, 10) imprime el número 10, pero con un ancho de cinco dígitos (es decir, rellenará con 3 espacios en blanco a la izquierda).
Precision	**Descripción**
número	Tamaño de la parte decimal para números reales. Número de caracteres a imprimir para cadenas de texto
*	Igual que el caso anterior, pero el número a utilizar se pasa como parámetro justo antes del valor. Por ejemplo printf("%.*s", 3, "abcdef")imprime "abc"
Length	**Descripción**
hh	Convertir variable de tipo char a entero e imprimir
h	Convertir variable de tipo short a entero e imprimir
l	Para enteros, se espera una variable de tipo long.
ll	Para enteros, se espera una variable de tipo long long.
L	Para reales, se espera una variable de tipo long double.
z	Para enteros, se espera un argumento de tipo size_t.
Type	**Descripción**
%c	Imprime el carácter ASCII correspondiente
%d, %i	Conversión decimal con signo de un entero
%x, %X	Conversión hexadecimal sin signo
%p	Dirección de memoria (puntero)
%e, %E	Conversión a coma flotante con signo en notación científica
%f, %F	Conversión a coma flotante con signo, usando punto decimal
%g, %G	Conversión a coma flotante, usando la notación que requiera menor espacio
%o	Conversión octal sin signo de un entero
%u	Conversión decimal sin signo de un entero
%s	Cadena de caracteres (terminada en '\0')
%%	Imprime el símbolo %

CLASES

```
[visibilidadClase] [modificadoresClase] class <NombreClase> [extends <NombreClasePadre>]
{
        // declaración de atributos (campos)
        visibilidad[modificadores] tipo nombreAtributo1 [= valor];
        visibilidad [modificadores] tipo nombreAtributo2 [= valor];
        ...
        // constructor
        public NombreClase(parámetros) {
                intrucciones;
        }
        ...
        // declaración de métodos
        visibilidad [modificadores] tipo nombreMetodo1(argumentos) {
                instrucciones;
        }
}
```

- **Dónde:**
  - **NombreClase:** nombre de la clase(1ª letra de cada palabra en mayúsculas)
  - **visibilidadClase**: posibles valores: **public**. Si no se especifica nada, esta clase solo es visible desde las clases del paquete donde esté.
  - **modificadoresClase:**
    - **final**: no podrás crear clases que hereden de ella
    - **abstract:** una clase que no implementa ningún método
  - **Nombre atributos y métodos**: 1er carácter del nombre en minúsculas, después cada palabra: 1er carácter en mayúsculas (Ej: numerPrimo, buscarEmpleado,etc..)
  - **visibilidad:** posibles valores: **public**/**protected** /**private**
    - **Por defecto (no poner nada)**: es visible desde las todas las clases del paquete donde esté
    - **public:** Indica que el atributo o método es visible desde cualquier clase.
    - **protected:** Indica que el atributo o método es visible desde cualquier otra clase del mismo paquete y clases heredadas
    - **private:** indica que el método o atributo sólo es visible en la clase donde está definido.
  - **Modificadores de atributos:** posibles valores: **final** /**static**
    - **final**: un atributo que no se puede modificar (constante)
    - **static**: un atributo que su valor es el mismo para todos los objetos instanciados de esa clase.
  - **Modificadores de atributos:** posibles valores: **final** /**static** /**abstract**
    - **final**: método: no permite ser sobrescrito por las clases descendientes.
    - **static**: es un método que se puede invocar sin necesidad decrear un objeto de la clase.
    - **abstract:** método que no tiene implementación en esta clase, se realizará en las clases descendientes.
    - **synchronized:** el entorno de ejecución obligará a que cuando un proceso esté ejecutando ese método, el resto de procesos que tengan que llamar a ese mismo método deberán esperar a que el otro proceso termine.
  - **argumentos**: declaración de variables separadas por comas

CONSTRUCTOR

- Se ejecuta automáticamente cuando se declara un objeto de una clase
- Tiene el mismo nombre que la clase. Una clase puede tener más de un constructor (sobrecarga).
- Por defecto, toda clase tiene un constructor sin parámetros y sin código, que desaparece una vez se escribe un método o el constructor para dicha clase.
- Por defecto, en las clases con herencia antes de ejecutarse todo constructor, llama al código del constructor sin parámetros de la clase padre. Se puede cambiar escribiendo como primera línea de código del constructor: super(parámetros).

OBJETO

- Declaración
  <NombreClase> <nombreObjeto>;
  <nombreObjeto> = new <NombreClase>(parámetros_constructor);
  <NombreClase> <nombreObjeto> = new <NombreClase> (parámetros_constructor)
- Acceso a un **atributo**
  nombreObjeto.atributo
- Llamada a un **método** de la clase de objeto
  nombreObjeto.metodo(argumentos);

PROGRAMA PRINCIPAL EN UNA CLASE JAVA

- Es aquella clase que contenga el método **main**(), que será el convierte a una clase en ejecutable:

```
class <Nombre>{
        public static void main (String[] args){
                instrucciones;
        }
}
```

LIBRERIAS

- Al principio del fichero de la clase, se debe escribir el paquete dónde se insertará la clase:
  package carpeta.subcarpeta. ...;
- Para utilizar código de otro fichero:
  import carpeta.subcarpeta.Nombreclase;
- Ejemplo:

```
import modelo.Nomina;
import java.lang.Math;
import java.io.*;
```

SENTENCIAS DE CONTROL

- Una instrucción: puede ocupar varias líneas y acaba en punto y coma: ";":
  instrucción;
- Los bloques de instrucciones van entre llaves:
  { instrucción1; instrucción2;...}

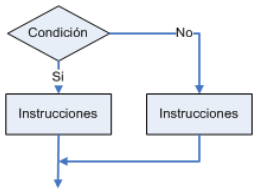
SENTENCIAS CONTROL FLUJO EJECUCIÓN

Tipo Sentencia	Sentencia en Java
Condicional simple y compuesta	**if** <condición> {...} [**else** {...}]
Condicional múltiple	**switch** (expresión) **case** <valor>: .... **break**;
Iterativa con condición inicial	**while** <condición> {...}
Iterativa final condición final	**do** {...} **while** <condición>
Repetitiva	**for** (inic, comp,incr) {...}
Otras	**break, continue, label, return <valor>, exit**

- **Sentencia de Selección o Condicional simple y compuesta**

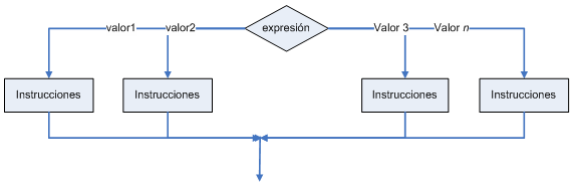
if (condición**) {**
  instrucciones1**;**
}
[else {
  Instrucciones2**;**
}]

if (condición**) {**
  instrucciones1**;**
}
[else {
  Instrucciones2**;**
}]



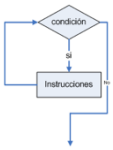
- **Sentencia de Selección o Condicional múltiple**

switch (expresión) {
  case <valor>: instrucciones; [**break**;]
  case <valor>: instrucciones; [**break**;]
  ...
  [**default**: instrucciones;]
}



- **Iterativa con condición inicial while (condición)**

while (condición){
  instrucciones;
}



- **Iterativa con condición final**

do {
  instrucciones;
} **while** (condición)

- **Repetitiva**

for (inicialización; comparación; incremento){
  Instrucciones;
}

EXCEPCIONES

- Permiten controlar posibles situaciones de error

```
try {
        código donde se pueden producir excepciones
}
catch (TipoExcepcion1 NombreExcepcion) {
        Código a ejecutar si se produce una excepción del tipo TipoExcepcion1
}
catch (TipoExcepcion2 NombreExcepcion) {
        Código a ejecutar si se produce una excepción del tipo TipoExcepcion1
}
...
finally {
        Código a ejecutar tanto si se produce una excepción como si no
}
```

- Java posee multitud de excepciones agrupadas por familias

ArithmeticException	**IOException**
EOFException	**FileNotFoundException**
NullPointerException	**NegativeArraySizeException**
ArrayIndexOutOfBoundsException	**SecurityException**
NumberFormatException, ...	

- Java posee multitud de "errores"
- Fallos de la máquina virtual que es mejor que no los gestione la aplicación.

throws:

- Con esta opción se está indicando que el método que lleve **throws** puede provocar excepciones que serán capturadas por los métodos que lo llamen.
- Ejemplo:

```
void ejemploExcepciones() throws InputMismatchException{
        ...
}

Cuando se llama al método hay que capturar la excepción:
        try{
                ...
                objeto. ejemploExcepciones();//puede saltar la  excepción
                ...
        }
        catch(InputMismatchException e){
                System.out.println("Tecleado caracteres erróneos");
        }
```

throw:

- La sentencia `throw` se utiliza para lanzar explícitamente una excepción.
- Ejemplo:

```
private static int metodoExcepcion() throws InputMismatchException {
  try {
        ....
  } catch (InputMismatchException e) {
      throw new InputMismatchException("Caracteres erróneos");
  }
}
Para llamar al método:
  try {
        codigo = metodoExcepcion();
  } catch (InputMismatchException e) {
      System.out.println(e.getMessage());
   }
```

SOBRECARGA

- Dos o más métodos con el mismo nombre dentro de la misma clase.
- Se diferencian en los parámetros

```
class nombre_clase {
      public tipo_retorno nombre_método(parámetros) {
            instrucciones;
      }
      public tipo_retorno nombre_método(otros parámetros) {
            otro instrucciones;
      }
      ...
}
```

this

- Se utiliza como referencia del objeto actual que está ejecutando el método. Es útil para diferenciar los atributos de la clase de los parámetros, cuando éstos tengan el mismo nombre y también cuando haya que llamar a un método pasando como referencia el objeto actual que está ejecutando el código.

HERENCIA

- Una clase puede heredar o derivar de otra: Pasa a disponer automáticamente de todos los métodos y atributos de esta otra clase como si fueran propios. La clase que hereda se llama "hija" y la clase de la cual hereda se llama "padre".

- Por defecto, toda clase hereda de la clase Object, a menos que especifiquemos que hereda de otra clase mediante la palabra extends:

```
class clase_hija extends clase_padre {
...
}
```

- La clase hija puede redefinir los métodos heredados de la clase padre. Un método de la clase hija puede llamar al código de un método de la clase padre mediante la palabra super:

```
tipo_retorno metodo_clase_hija(argumentos) {
... super. metodo_clase_padre(argumentos) ... //llamada método clase padre
...
}
```

- Un objeto de la clase hija es del tipo de la clase hija, pero también del tipo de la clase padre y del tipo de todos sus antecesores.
  Operador lógico **instanceof**: Permite preguntar por el tipo de un objeto
  objeto **instanceof** nombre_clase
- Se puede cambiar el tipo de un objeto escribiendo previamente su nuevo tipo entre paréntesis:
  (nueva_clase) objeto
- Una clase sólo puede heredar de otra clase y de varias interfaces. En Java no existe la herencia múltiple.

VISIBILIDAD

- De los métodos y los atributos:
  - **Por defecto (no poner nada)**: es visible desde las todas las clases del paquete donde esté
  - **public:** Indica que el atributo o método es visible desde cualquier clase.
  - **protected:** Indica que el atributo o método es visible desde cualquier otra clase del mismo paquete y clases heredadas
  - **private:** indica que el método o atributo sólo es visible en la clase donde está definido.

- **Modificadores de atributos:** posibles valores: **final** /**static**
  - **final**: un atributo que no se puede modificar. Se utiliza para declarar una **constante:** `final double PI = 3.141592;`
  - **static**: un atributo que su valor es el mismo para todos los objetos instanciados de esa clase. El valor de un atributo estático es compartido por todos los objetos de dicha clase

- **Modificadores de métodos:** posibles valores: **final** /**static** /**abstract**
  - **final**: método que no puede ser sobrescrito por las clases descendientes.
  - **static**: es un método que se puede invocar sin necesidad decrear un objeto de la clase.
  - **abstract:** método que no tiene implementación en esta clase, se realizará en las clases descendientes. Las clases con métodos abstractos no se pueden instanciar y sus clases herederas deberán escribir el código de sus métodos abstractos si se quiere crear alguna instancia suya
  - **synchronized:** el entorno de ejecución obligará a que cuando un proceso esté ejecutando ese método, el resto de procesos que tengan que llamar a ese mismo método deberán esperar a que el otro proceso termine.

equals y clone

Variables simples

- El nombre de una variable de tipo simple indica la dirección de memoria que contiene el valor de la variable (referencia directa).
  int a, b;
- Operador "==": permite comparar valores de variables simples
  ... (a == 3)...
  ... (a == b)...

- Operador "=": permite asignar un valor a una variable simple
  a = 3;
  a = b;

Objetos

- El nombre de un objeto no contiene los valores de los atributos, sino la posición de memoria donde residen dichos valores de los atributos (referencia indirecta)
  Complejo w, z;
- Operador == : permite comparar si dos objetos ocupan la misma posición (comprueba si son el mismo objeto)
  ... (w == z)...
- Método **equals** : permite comprobar si dos objetos poseen atributos con los mismos valores (compara los contenidos)w.equals (z)
- Operador = : "asigna" un objeto a otro objeto que ya existe (serán el mismo objeto)
  Complejo w, z;
  w = z;
- A partir de este instante, "w" ocupa la misma posición de memoria que "z".
- Método **clone**: crea una copia de un objeto determinado y la asigna a otro.
  Complejo w = (Complejo) z.clone ();

POLIMORFISMO

- Se puede declarar un objeto de una clase, pero instanciarlo como un descendiente de dicha clase lo contrario no es posible

  clase_padre objeto = new clase_descendiente(parámetros_constructor);

- Ejemplo:

```
public class Complex {
      double re, im;

      public Complex(double re, double im) {
            this.re = re;
            this.im = im;
      }

      public void imprimir() {
            System.out.println(re + " " + im);
      };
}
public class Complex1 extends Complex {
    public Complex1(double re, double im) {
            super(re, im);
    }
    @Overide
    public void imprimir() {
            System.out.println(re + "+" + im +
    "i");
    }
 }
```

```
 public class Complex2 extends Complex {
       public Complex2(double re, double im) {
             super(re, im);
       }
       @Overide
       public void imprimir() {
             System.out.println("(" + re + "," +
       im + ")");
       }
 }
public class Principal {
       public   static   void   main(String[]
              args){
              Complex v[] = new Complex[2];
              v[0] = new Complex1(5, 4);
              v[1] = new Complex2(1, 3);
              for (int i = 0; i < v.length;
                     i++) {
                     v[i].imprimir();
              }
       }
}
```

ALMACENANDO DATOS. FICHEROS

CLASE FILE

Constructor	Descripción
public **File** (String ruta_absoluta)	Crea un objeto File al cual hay que pasarle el nombre del archivo
public **File** (String Ruta, String Nombre)	Se le pasa como primer parámetro la ruta relativa y el nombre del fichero
public **File** (File ruta, String Nombre)	El objeto File puede representar un directorio o un fichero, el primero es la ruta y el segundo el nombre del fichero

Método	Función
String **getName**()	Devuelve el nombre del fichero o directorio.
String **getPath**()	Devuelve el camino relativo.
String **getAbsolutePath**()	Devuelve el camino absoluto del fichero/directorio.
boolean **canRead**()	Devuelve true si el fichero se puede leer
boolean **canWrite**()	Devuelve true si el fichero se puede escribir.
Long **length**()	Nos devuelve el tamaño del fichero en bytes.
boolean **createNewFile**()	crea un nuevo fichero, vacío, asociado a File si y solo si no existe un fichero con dicho nombre
String[] **list**()	Devuelve un array con los ficheros contenidos en el directorio
boolean **delete**()	Borra el fichero o directorio asociado al File.
boolean **exists**()	Devuelve true si el fichero/directorio existe.
String **getParent**()	Devuelve el nombre del directorio padre, o null si no existe.
boolean **isDirectory**()	Devuelve true si el objeto File corresponde a un directorio.
boolean **isFile**()	Devuelve true si el objeto File corresponde a un fichero normal.
boolean **mkdir**()	Crea un directorio con el nombre indicado en la creación de objeto File.
boolean **renameTo**(File n_nombre)	Renombra el fichero con n_nombre

FLUJOS O STREAMS: TIPOS

- **Flujos de caracteres** (16 bits) : Clases : **Reader** y **Writer**
- **Flujos de bytes** (8 bits) : Clases : **InputStream** y **OutputStream**.

FLUJOS DE CARACTERES:

- **Buferización** de datos: **BufferedReader** y **BufferedWriter**, se utilizan para evitar que cada lectura o escritura acceda directamente al fichero, ya que utilizan un buffer intermedio entre la memoria y el stream.

FLUJOS DE BYTES

- **FileInputStream:** manipulan los flujos de bytes provenientes de ficheros en disco
- **FileOutputStream:** manipulan los flujos de bytes dirigidos hacia ficheros en disco.

CLASE FileReader (Leer caracteres de un fichero)

- Constructores:
  FileReader(File fichero); **FileReader(String fichero);**

Método	Función
int read();	Lee un carácter y lo devuelve.
int read(char[] buf);	Lee hasta la máxima longitud de una matriz de caracteres (buf). Los caracteres leídos del fichero se van almacenando en el array **buf.**
int read(char[] buf, int desplazamiento,int n)	Lee hasta **n** caracteres de datos de la matriz **buf** comenzando por buf[desplazamiento J y devuelve el número leído de caracteres

- Devuelven el o los caracteres leídos o **-1** si se ha llegado al final del fichero

CLASE FileWriter (Escribir caracteres en un fichero)

- Constructores:
  FileWriter(File fichero[,true]); **FileWriter(String fichero,[true]);**

Método	Función
void **write(int c)**	escribe un carácter.
void **write(char[] buf)**	escribe un array de caracteres.
void **write(char[] buf, int desplazamiento, int n)**	escribe n caracteres de datos en la matriz comenzando por buf[desplazamiento].
void **write(String str)**	escribe una cadena de caracteres
append(char c)	añade un carácter a un fichero

- Estos métodos también pueden lanzar la excepción **IOException**.
- Para añadir caracteres al final del fichero: **FileWriter fic = new FileWriter (fichero, true) ;**

CLASE BufferedReader y BufferedWriter.

- Para construir un BufferedReader necesitamos la clase FileReader:
  BufferedRedader fichero = new BufferedReader
  (new FileReader (NombreFichero)) ;
- O también:
  FileReader f=new **FileReader (**NombreFichero**);**
  BufferedRedader fichero = new **BufferedReader (f);**

- **readLine*():*** lee líneas completas
- Para construir un **BufferedWriter** necesitamos la clase **FileWriter**:

BufferedWriter fichero = new BufferedWriter(new FileWriter(NombreFichero);

LA CLASE PrintWriter.

- La clase **PrintWriter**, también deriva de Writer: graba un String en un fichero.
- Métodos **print**(String) y **println**(string)

MÉTODOS DE LA CLASE FileInputStream (Leer en un fichero binario):

Método	Función
int read()	lee un byte y lo devuelve.
int read(byte[] b);	lee hasta b.length bytes de datos de una matriz de bytes
int read(byte[] b, int desplazamiento, int n)	lee hasta n bytes de la matriz b comenzando por b[desplazamiento] y devuelve el número leído de bytes.

- Devuelven el o los bytes leídos o -1 si se ha llegado al final del fichero

MÉTODOS DE LA CLASE FileOutputStream (Escribir en un fichero binario):

Método	Función
void **write(int c)**	escribe un byte.
void **write(byte[] b)**	escribe b.length bytes.
void **write(char[] buf, int desplazamiento, int n)**	escribe n bytes a partir de la matriz de bytes de entrada y comenzando por b[desplazamiento]

- Estos métodos también pueden lanzar la excepción **IOException**.
- Para añadir bytes al final: **FileOutputStream fileout = new FileOutputStream (fichero, true) ;**

LEER Y ESCRIBIR DATOS PRIMITIVOS

- Usaremos las clases:
  - **DataInputStream**
  - **DataOutputStream.**
- Estas clases tienen los métodos **read**() y **write**().

- Además:

Método para lectura	Métodos para escritura
Boolean **readBoolean**(); Byte **readByte**(); int **readUnsignedByte**(); int **readUnsignedShort**(); short **readShort**(); char **readChar**(); int **readInt**(); long **readLong**(); float **readFloat**(); double **readDouble**(); String **readUTF**();	void **writeBoolean** (boolean v); void **writeByte** (int v); void **writeBytes** (String s); void **writeShort** (int v); void **writeChars** (String s); void **writeChar** (int v); void **writeInt** (int v); void **writeLong** (long v); void **writeFloat** (float v); void **writeDouble** (double v); void **writeUTF** (String str);

- Para abrir un objeto DataInputStream, se utilizan los mismos métodos que FileInputStream, ejemplo:

File fichero **= new File ("C: \\EJERCICIOS\ \UNI1 \\FichData. dat");**
FileInputStream filein **= new FileInputStream (**fichero**) ;**
DataInputStream dataIS **= new DataInputStream (**filein**) ;**

- Para abrir un objeto DataOutputStream, se utilizan los mismos métodos que FileOutputStream, ejemplo:

File fichero **= new File ("C: \\EJERCICIOS\ \UNI1 \\FichData. dat");**
FileOutputStream filein **= new FileOutputStream (**fichero**) ;**
DataOutputStream dataIS **= new DataOutputStream (**filein**) ;**

OBJETOS SERIALIZABLES.

- **ObjectInputStream** : clase para leer objetos serializables en un stream
- **ObjectOutputStream** : clase para escribir objetos serializables en un stream

Flujo de Entrada
ObjectInputStream FinPersona=new **ObjectInputStream** (new FileInputStream ("FichPersona.dat"));
File fichero=new File("FichPersona.dat") ; **ObjectInputStream** FinPersona=new **ObjectInputStream** (new FileInputStream (fichero));

Flujo de Salida
ObjectOutputStream foutPersona=new **ObjectOutputStream** (new FileOutputStream ("FichPersona.dat"));
File fichero=new File("FichPersona.dat") ; **ObjectOutputStream** foutPersona=new **ObjectOutputStream** (new FileOutputStream (fichero));
File fichero=new File("FichPersona.dat") ; **ObjectOutputStream** foutPersona=new **MiObjectOutputStream** (new FileOutputStream (fichero,true)); Cuando el fichero existe, para añadir datos hay que utilizar una clase

Método para lectura	Métodos para escritura
object readObject()	**void writeObject(Object)**

- **Excepciones : IOException, ClassNotFound Exception**
- **Excepción : IOException;**

FICHEROS DE ACCESO ALEATORIO.

- **Crear** un fichero de acceso aleatorio:
  - Escribiendo el nombre del fichero:

    fichero = new RandomAccessFile(String nombre, String modoAcceso);

  - Con un objeto File:

    fichero = new RandomAccessFile(File fich, String modoAcceso);

  - Modos de acceso: r = lectura, rw: lectura/escritura,

- **Métodos** que manejan el puntero:

  - *long* ***getFilePointer()***: devuelve la posición actual del puntero del fichero
  - void **seek(long 19osición)**: coloca el puntero del fichero en una posición determinada desde el comienzo del mismo.
  - long **length()**: devuelve el tamaño del fichero en bytes. La posición lengthO marca el final del fichero.
  - int **skipBytes(int desplazamiento)**: desplaza el puntero desde la posición actual el número de bytes indicados en desplazamiento.

Tipo de datos	N º de bytes que ocupa	Tipo de datos	N º de bytes que ocupa
short	2	**double**	8
int	4	**char**	2
float	4	**boolean**	1

APLICACIÓN DE LAS ESTRUCTURAS DE ALMACENAMIENTO

CADENAS DE CARACTERES.

- clase **String**: String cadena="Hola";

OPERACIONES AVANZADAS CON CADENAS DE CARACTERES

Método	Descripión
concatenación	`String cad = "¡Bien "+"venido!";` `System.out.println(cad); // cad tiene "!Bien venido!`
int length()	Longitud de una cadena: int l=cad.length(); // l tendrá 12

char charAt(int pos)	Retorna el carácter ubicado en la posición pasada por parámetro.
	`char t = cad.charAt(6);`
	`System.out.println(t); // t tendrá "v"`

String substring(int beginIndex, int endIndex).	Extraer una subcadena de otra de mayor tamaño. `String s=cad.substring(0,5); // Devolvería: "!Bien".`
String substring (int beginIndex)	`String subcad = cad.substring(2);` `System.out.println(subcad); // subcad tendrá "!B"`

Integer.valueOf(), Long.valueOf(), Short.valueOf(), Float.valueOf(), Double.valueOf ()	Convertir cadenas a números
	`String c="1234.5678";`
	`double n;`
	`try {n=`**`Double.valueOf`**`(c).doubleValue();`
	`} catch (NumberFormatException e)`

cad1.compareTo(cad2)	Permite comparar dos cadenas entre sí lexicográficamente. Retornará 0 si son iguales, un número menor que cero si la cadena (cad1) es anterior en orden alfabético a la que se pasa por argumento (cad2), y un número mayor que cero si la cadena es posterior en orden alfabético.
cad1.equals(cad2)	Cuando se comparan si dos cadenas son iguales, no se debe usar el operador de comparación "==", sino el método equals. Retornará true si son iguales, y false si no lo son.
cad1.compareToIgnoreCase(cad2) **cad1.equalsIgnoreCase(cad2)**	El método compareToIgnoreCase funciona igual que el método compareTo, pero ignora las mayúsculas y las minúsculas a la hora de hacer la comparación. Las mayúsculas van antes en orden alfabético que las minúsculas, por lo que hay que tenerlo en cuenta. El método equalsIgnoresCase es igual que el método equals pero sin tener en cuenta las minúsculas.
cad1.trim()	Genera una copia de la cadena eliminando los espacios en blanco anteriores y posteriores de la cadena.
cad1.toLowerCase()	Genera una copia de la cadena con todos los caracteres a minúscula.
cad1.toUpperCase()	Genera una copia de la cadena con todos los caracteres a mayúsculas.
cad1.indexOf(cad2) **cad1.indexOf(cad2,num)**	Si la cadena o carácter pasado por argumento está contenida en la cadena invocante, retorna su posición, en caso contrario retornará -1. Opcionalmente se le puede indicar la posición a partir de la cual buscar, lo cual es útil para buscar varias apariciones de una cadena dentro de otra.
cad1.contains(cad2)	Retornará true si la cadena pasada por argumento está contenida dentro de la cadena. En caso contrario retornará false.
cad1.startsWith(cad2)	Retornará true si la cadena comienza por la cadena pasada como argumento. En caso contrario retornará false.
cad1.endsWith(cad2)	Retornará true si la cadena acaba por la cadena pasada como argumento. En caso contrario retornará false.
cad1.replace(cad2,cad3)	Generará una copia de la cadena cad1, en la que se reemplazarán todas las apariciones de cad2 por cad3. El reemplazo se hará de izquierda a derecha

- **StringBuilder** : Permite modificar la cadena que contiene, no crea una nueva instancia.

```
StringBuilder strb=new StringBuilder ("Hoal Muuundo");
```

  - **strb.delete(6,8);** Eliminamos las 'uu' que sobran en la cadena.
  - **strb.append ("!");** Añadimos al final de la cadena el símbolo de cierre de exclamación.
  - **strb.insert (0,"¡");** Insertamos en la posición 0, el símbolo de apertura de exclamación.
  - **strb.replace (3,5,"la");** Reemplazamos los caracteres 'al' situados entre la posición inicial 3 y la posición final 4, por la cadena 'la'.

EXPRESIONES REGULARES

Patrón	Significado
[xyz]	Entre corchetes podemos indicar opcionalidad. Solo uno de los símbolos que hay entre los corchetes podrá aparecer en el lugar donde están los corchetes.
"[a-z]" **"[A-Z]"** **"[a-zA-Z]"** **"[0-9]"**	Usando el guión y los corchetes podemos indicar que el patrón admite cualquier carácter entre la letra inicial y la final. "[a-z]": admite 1 letra minúscula "[A-Z]": admite 1 letra mayúscula "[a-zA-Z]": admite 1 letra en minúscula o en mayúscula "[0-9]": un dígito numérico entre 0 y 9
"a?"	Usaremos el interrogante para indicar que un símbolo puede aparecer **una vez o ninguna**. De esta forma la letra "a" podrá aparecer una vez o simplemente no aparecer.**(0,1)**
"a*"	Usaremos el asterisco para indicar que un símbolo puede aparecer **una vez o muchas veces**, pero también ninguna.**(0,N)**
"a+"	Usaremos el símbolo de suma para indicar que otro símbolo debe aparecer **al menos una vez**, pudiendo repetirse cuantas veces quiera**.(1,N)**
"a{1,4}"	Usando las llaves, podemos indicar el número **mínimo y máximo** de veces que un símbolo podrá repetirse. Posibilidades: a, aa, aaa, aaaa.
"a{2,}"	También es posible indicar solo el número **mínimo** de veces que un carácter debe aparecer (sin determinar el máximo), haciendo uso de las llaves, indicando el primer número y poniendo la coma (no la olvides).
"a{5}".	A diferencia de la forma anterior, si solo escribimos un número entre llaves, sin poner la coma detrás, significará que el símbolo debe aparecer un **número exacto de veces**. En este caso, la "a" debe aparecer exactamente 5 veces.
"[a-z]{1,4}[0-9]+"	Los indicadores de repetición se pueden usar también con corchetes, dado que los corchetes representan, básicamente, un símbolo. En el ejemplo anterior se permitiría de una a cuatro letras minúsculas, seguidas de al menos un dígito numérico.
"[^abc]"	El símbolo "^", cuando se pone justo detrás del corchete de apertura, significa **"negación".** La expresión regular admitirá cualquier símbolo diferente a los puestos entre corchetes. En este caso, cualquier símbolo diferente de "a", "b" o "c".
"^[01]+$"	Cuando el símbolo **"^"** aparece al comienzo de la expresión regular, permite indicar **comienzo de línea** o de entrada. El símbolo **"$"** permite indicar **fin de línea** o fin de entrada. Util cuando se trabaja en modo multilínea y con el método find().
"."	El punto simboliza cualquier carácter.
"\\d"	**Un dígito** numérico. Equivale a "[0-9]".
"\\D"	Cualquier cosa **excepto un dígito** numérico. Equivale a "[^0-9]"
"\\s"	Un espacio en **blanco** (incluye tabulaciones, saltos de línea y otras formas de espacio).
"\\S"	Cualquier cosa **excepto** un espacio en **blanco**.
"\\w"	carácter que podrías encontrar en una palabra. Equivale a "[a-zA-Z0-9]".

- Cuando hay varias posibles cadenas válidas, en la expresión regular se ponen entre paréntesis y separadas por **|**.
  `Pattern p=Pattern.compile("[0-9]{1,2}|A[0-9]{2}");`
  Puesde ser: 1 o 2 dígitos numéricos o la letra A seguida de 2 dígitos numéricos

- Java ofrece las clases Pattern y Matcher contenidas en el paquete java.util.regex.*.
  - La clase **Pattern** se utiliza para procesar la expresión regular y "compilarla", lo cual significa verificar que es correcta y dejarla lista para su utilización.
  - La clase **Matcher** sirve para comprobar si una cadena cualquiera sigue o no un patrón.

- Veámoslo con un ejemplo:

```
Pattern p=Pattern.compile("[01]+");
Matcher m=p.matcher("00001010");
if (m.matches())
      System.out.println("Si, contiene el patrón");
else
      System.out.println("No, no contiene el patrón");
```

- La clase **Matcher** contiene el resultado de la comprobación y ofrece varios métodos para analizar la forma en la que la cadena ha encajado con un patrón:
  - **m.matches().** Devolverá true o false si toda la cadena (de principio a fin) encaja con el patrón
  - **m.lookingAt().** Devolverá true si el patrón se ha encontrado al principio de la cadena
  - **m.find().** Devolverá true si el patrón existe en algún lugar la cadena (no necesariamente toda la cadena debe coincidir con el patrón). Para obtener la posición exacta donde se ha producido la coincidencia con el patrón podemos usar los métodos **m.start() y m.end()**, para saber la posición inicial y final donde se ha encontrado. Una segunda invocación del método find() irá a la segunda coincidencia (si existe), y así sucesivamente. Podemos reiniciar el método find(), para que vuelva a comenzar por la primera coincidencia, invocando el método **m.reset()**.
- Los **paréntesis**: definen **grupos**, por ejemplo: "(#[01]){2,3}, admitiría: "#0#1" o "#0#1#0".

CREACIÓN DE ARRAYS

- Los **arrays** permiten almacenar una colección de objetos o datos del mismo tipo.
- Declaración de un array de una dimensión:
  <tipo_datos> <nombre_array>[];
  <nombre_array> = new <tipo_datos>[<num_elementos>];
- Creación del array: **nombre=new tipo[dimension];**

```
int[] n;                 // Declaración del array.
n = new int[10];         //Creación del array reservando para el un tamaño de 10
int[] m=new int[10];     // Declaración y creación en un mismo lugar.
```

- Acceso a un valor: `int suma=Numeros[0] + Numeros[1] + Numeros[2];`
- Propiedad **length** nos permite saber el tamaño de cualquier array:

Inicialización.

- Ejemplo de inicialización:

```
int[] array = {10, 20, 30};
String[] diassemana={"Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado",
"Domingo"};
```

- La inicialización anterior funciona cuando se trata de un tipo de dato primitivo (int, short, float, double, etc.) o un String, y algunos pocos casos más, pero no funcionará para cualquier objeto.

ARRAYS MULTIDIMENSIONALES

- Declaración de un array de una varias dimensiones:
  <tipo_datos> <nombre_array>[][]...[] ;
  <nombre_array> = new <tipo_datos> [<n1]...[<nk>];
- La forma de crear un array de dos dimensiones en Java es la siguiente:
  `int[][] a2d=new int[4][5];`
- Inicializarlos:
  `int[][] a2d={{0,1,2},{3,4,5},{6,7,8},{9,10,11}};`
  `int[][][] a3d={{{0,1},{2,3}},{{0,1},{2,3}}};`

INTRODUCCIÓN A LAS COLECCIONES (Colletion)

- Las colecciones almacenan y manipulan grupos de objetos que, a priori, están relacionados entre sí
- Operaciones más importantes definidas por la interfaz **java.util.Collection**, "<E>" es el parámetro de tipo (podría ser cualquier clase):

Método de Colletion	Significado
int **size**()	Retorna el número de elementos de la colección.
boolean **isEmpty**()	Retornará verdadero si la colección está vacía
boolean **contains** (Object element)	Retornará verdadero si la colección tiene el elemento pasado como parámetro
boolean **add**(E element)	Añadir elementos a la colección
boolean **remove** (Object element)	Eliminar elementos de la colección
Iterator<E> iterator()	Crear un iterador para recorrer los elementos de la colección.
Object[] **toArray**()	Pasa la colección a un array de objetos tipo Object
containsAll(Collection<?> c)	Comprueba si una colección contiene los elementos existentes en otra colección, si es así, retorna verdadero
addAll (Collection<? extends E> c)	Añade todos los elementos de una colección a otra colección, siempre que sean del mismo tipo (o deriven del mismo tipo base)
boolean removeAll(Collection<?> c)	Si los elementos de la colección pasada como parámetro están en nuestra colección, se eliminan, el resto se quedan
boolean retainAll(Collection<?> c)	Si los elementos de la colección pasada como parámetro están en nuestra colección, se dejan, el resto se eliminan
void clear()	Vacía la colección

CONJUNTOS

- Los **conjuntos** : **no admite duplicados**.
- La interfaz **java.util.Set** (extiende de la interfaz Collection).
- Las implementaciones (clases genéricas que implementan la interfaz Set) más usadas son las siguientes:
  - **java.util.HashSet.** Conjunto que almacena los objetos usando tablas hash, **acelera** enormemente el acceso a los objetos almacenados.
  - **java.util.LinkedHashSet.** Conjunto que almacena objetos combinando tablas hash, para un acceso **rápido** a los datos, y listas enlazadas para conservar el **orden**.
  - **java.util.TreeSet.** Conjunto que almacena los objetos usando unas estructuras conocidas como árboles rojo-negro. Los datos almacenados se **ordenan por valor.**

```
HashSet<Integer> conjunto=new HashSet<Integer>();
Integer n=new Integer(10);
if (!conjunto.add(n)) System.out.println("Número ya en la lista.");
```

- **Iteradores**:
```
for (Integer i: conjunto)
      { System.out.println("Elemento almacenado:"+i); }
```
- **Operaciones de conjuntos:**
```
TreeSet<Integer> A= new TreeSet<Integer>();
// Elementos del conjunto A: 9,19,5 y 7
A.add(9); A.add(19); A.add(5); A.add(7);
LinkedHashSet<Integer> B= new LinkedHashSet<Integer>();
// Elementos del conjunto B: 10, 20, 5 y 7
B.add(10); B.add(20);B.add(5);B.add(7);
```

A.addAll(B)	UNION: Todos los del conjunto A, añadiendo los del B, pero sin repetir los que ya están: 5, 7, 9, 10, 19 y 20.
A.removeAll(B)	A-B: Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19.
A.retainAll(B)	INTERSECCIÓN: Todos los elementos del conjunto A, que también están en el conjunto B: 5 y 7.

- Los **TreeSet** tienen un conjunto de operaciones adicionales, es útil cuando el tipo de objeto que se almacena no es un simple número, sino algo más complejo (un artículo, por ejemplo).
- **TreeSet** es capaz de ordenar tipos básicos (números, cadenas y fechas) pero otro tipo de objetos no puede ordenarlos con tanta facilidad.
- Para que los conjuntos de objetos no admitan duplicados, para que los ordene, tenemos que modificar los métodos equals, hashCode y compare Producto y que implemente: compare
- Para ordenar los elementos, debemos decirle cuándo un elemento va antes o después que otro, y cuándo son iguales. Para ello, utilizamos la interfaz genérica **java.util.Comparator.**

```
class ComparadorDeObjetos implements Comparador<Objeto> {
   @Override
   compare(Objeto o1, Objeto o2) {
         int sumao1=o1.a+o1.b;
         int sumao2=o2.a+o2.b;
         if (sumao1<sumao2) return 1;
         else if (sumao1>sumao2) return -1;
         else return 0;
   }
}
```

LISTAS

- Las listas : amplían el conjunto de operaciones de las colecciones añadiendo operaciones:
  Las listas si pueden almacenar duplicados,Acceso posicional, Búsqueda, Extracción de sublistas.
- Se utiliza la intefaz llamada **java.util.List**.
- Dos implementaciones:
  java.util.LinkedList : doblemente enlazadas
  java.util.ArrayList

Método de List	Significado
E get(int index)	El método get permite obtener un elemento partiendo de su posición (index)
E set(int index, E element)	El método set permite cambiar el elemento almacenado en una posición de la lista (index), por otro (element)
void add(int index, E element).	Se añade otra versión del método add, en la cual se puede insertar un elemento (element) en la lista en una posición concreta (index), desplazando los existentes
E remove(int index)	Se añade otra versión del método remove, esta versión permite eliminar un elemento indicando su posición en la lista.
boolean addAll(int index, Collection<? extends E> c)	Se añade otra versión del método addAll, que permite insertar una colección pasada por parámetro en una posición de la lista, desplazando el resto de elementos
int indexOf(Object o)	El método indexOf permite conocer la posición (índice) de un elemento, si dicho elemento no está en la lista retornará -1
int lastIndexOf(Object o)	El método lastIndexOf nos permite obtener la última ocurrencia del objeto en la lista (dado que la lista si puede almacenar duplicados)
List<E> subList(int from, int to)	El método subList genera una sublista (una vista parcial de la lista) con los elementos comprendidos entre la posición inicial (incluida) y la posición final (no incluida)

- **<E>** corresponde con el tipo base usado como parámetro genérico al crear la lista.
- Para usar una lista haremos uso de sus implementaciones **LinkedList** y **ArrayList.**

```
//Declaración y creación del LinkedList de enteros.
LinkedList<Integer> t=new LinkedList<Integer>();
t.add(1); t.add(3); t.add(1,2); // Añade elementos a la lista
t.add(t.get(1)+t.get(2)); // Suma los valores pos. 1 y pos 2 y los añade al final
t.remove(0); // Elimina el primer elemento de la lista.
for (Integer i: t) System.out.println("Elemento:”+i);
```

- Veamos otro ejemplo, esta vez con ArrayList, de cómo obtener la posición de un elemento en la lista:

```
// Declaración y creación del ArrayList de enteros.
ArrayList<Integer> al=new ArrayList<Integer>();
al.add(10); al.add(11); // Añadimos dos elementos a la lista.
// Sustituimos el 11 por el 12, primero lo buscamos y luego lo reemplazamos.
al.set(al.indexOf(11), 12);
```

- Los **LinkedList** utilizan listas **doblemente enlazadas**, que son listas enlazadas (como se vio en un apartado anterior), pero que permiten ir hacia atrás en la lista de elementos.
- Si hay eliminaciones, lista enlazada: **LinkedList**
- Si no hay eliminaciones, si insertar y consultar :**ArrayList**.
- Los objetos inmutables no pueden ser modificados después de su creación, por lo que cuando se incorporan a la lista, a través de los métodos add, se pasan por copia (es decir, se realiza una copia de los mismos).
- En cambio, los objetos mutables (como las clases que tú puedes crear), no se copian, se pasa un puntero a los mismos.
- En la lista no se almacena una copia del objeto Test, sino un apuntador a dicho objeto

CONJUNTOS DE PARES CLAVE/VALOR.

- Tenemos por un lado cada palabra y por otro su significado. Para resolver este problema existen precisamente los arrays asociativos. Un tipo de array asociativo son los mapas o diccionarios, que permiten almacenar pares de valores conocidos como clave y valor. La clave se utiliza para acceder al valor, como una entrada de un diccionario permite acceder a su definición.
- En Java existe la interfaz **java.util.Map** que define los métodos que deben tener los mapas, y existen tres implementaciones principales de dicha interfaz:

  java.util.**HashMap**, java.util.**TreeMap** y java.util.**LinkedHashMap.**

```
HashMap<String,Integer> t=new HashMap<String,Integer>();
```

ITERADORES

- Con bucles for-each:
```
ArrayList<Integer> listaEnteros=new ArrayList <Integer>();
for (Integer i: listaEnteros) System.out.println("Elemento:”+i);
```
- Bucle normal creando un iterador:
```
Iterator<Integer> it=ListaEnteros.iterator();
// Mientras que haya un siguiente elemento, seguiremos en el bucle.
while (it.hasNext()){
  Integer t=it.next(); // Escogemos el siguiente elemento.
  if (t%2==0) it.remove();  // eliminamos los elementos con valores pares
}
```
- **Método 1º (mejor)**
```
ArrayList <Integer> lista=new ArrayList<Integer>();
for (int i=0;i<10;i++) lista.add(i);
Iterator<Integer> it=lista.iterator();
while (it.hasNext()) {
            Integer t=it.next();
            if (t%2==0) it.remove();
}
```
- **Método 2º**
```
ArrayList <Integer> lista=new ArrayList<Integer>();
for (int i=0;i<10;i++) lista.add(i);
Iterator it=lista.iterator();
while (it.hasNext()) {
            Integer t=(Integer)it.next();
            if (t%2==0) it.remove();
}
```

ALGORITMOS

Operación	Descripción	Ejemplos
Ordenar un array	Ordena un array	`Integer[] array={10,9,99,3,5};` `Arrays.sort(array);`

Ordenar una lista	Ordena una lista con números	`ArrayList<Integer> lista=new ArrayList<Integer>();`
		`lista.add(10); lista.add(9);lista.add(99);`
		`lista.add(3); lista.add(5);`
		`Collections.sort(lista);`

Desordenar una lista.	Desordena una lista, este método no está disponible para arrays.	`Collections.shuffle (lista);`
Rellenar una lista o array.	**Rellena** una lista o array copiando el mismo valor	`Collections.fill (lista,elemento);` `Arrays.fill (array,elemento);`
Búsqueda binaria.	Permite realizar búsquedas rápidas en un una lista o array ordenados	`Collections.binarySearch(lista,elemento);` `Arrays.binarySearch(array, elemento);`

Convertir un array a lista.	Convierte un **array** a una **lista** de elementos, retorna una lista que implementa la interfaz java.util.List.	`List lista=Arrays.asList(array);`
		Si el tipo de dato almacenado en el array es conocido (**Integer** por ejemplo), es conveniente especificar el tipo de objeto de la lista:
		`List<Integer>lista = Arrays.asList(array);`

Convertir una lista a array.	Convierte una **lista** a **array**.	Para este ejemplo, supondremos que los elementos de la lista son números, dado que hay que crear un array del tipo almacenado en la lista, y del tamaño de la lista:
		`Integer[] array=new Integer[lista.size()];`
		`lista.toArray(array)`

Dar la vuelta.	Pone una lista en orden inverso al que tiene.	`Collections.reverse(lista);`

- Imagina que tienes los artículos almacenados en una lista llamada "articulos", y que cada artículo se almacena en la siguiente clase (fíjate que el código de artículo es una cadena y no un número):

```
class Articulo {
    public String codArticulo; // Código de artículo
    public String descripcion; // Descripción del artículo.
    public int cantidad; // Cantidad a proveer del artículo.
}
```

- La 1ª forma de ordenar consiste en crear una clase que implemente la interfaz java.util.Comparator: `class comparadorArticulos implements Comparator<Articulo>`

```
{
    @Override
    public int compare( Articulo o1, Articulo o2) {
   return o1.codArticulo.compareTo(o2.codArticulo);
  }
}
```

- La 2ª forma: consiste en hacer que los objetos que se introducen en la lista o array implementen la interfaz java.util.Comparable.

```
class Articulo implements Comparable<Articulo>{
    public String codArticulo;
    public String descripcion;
    public int cantidad;

    @Override
    public int compareTo(Articulo o) {
   return codArticulo.compareTo(o.codArticulo); }
}
```

- Ordenar ahora la lista de artículos es sencillo, fíjate que fácil: **Collections.sort(articulos);**

FORMATEADO DE CADENAS EN JAVA.

Tipo de conversión	Especificación de formato	Tipos de datos aplicables	Ejemplo	Resultado del ejemplo
Valor lógico o booleano.	**"%b" o "%B"**	Boolean (cuando se usan otros tipos de datos siempre lo formateará escribiendo true).	boolean b=true; String d= String.format("Resultado: %b", b); System.out.println (d);	Resultado: true
Cadena de caracteres.	**"%s" o "%S"**	Cualquiera, se convertirá el objeto a cadena si es posible (invocando el método toString). "%-s": ajusta a la izquierda	String cad="hola mundo"; String d= String.format("Resultado: %s", cad); System.out.println (d);	Resultado: hola mundo
Entero decimal	**"%d"**	Un tipo de dato entero.	int i=10; String d= String.format("Resultado: %d", i); System.out.println (d);	Resultado: 10
Número en notación científica	**"%e" o "%E"**	Flotantes simples o dobles.	double i=10.5; String d= String.format("Resultado: %E", i); System.out.println (d);	Resultado: 1.050000E+01
Número decimal	**"%f"**	Flotantes simples o dobles.	float i=10.5f; String d= String.format("Resultado: %f", i); System.out.println (d);	Resultado: 10,500000
Número en notación científica o decimal (lo más corto)	**"%g" o "%G"**	Flotantes simples o dobles. El número se mostrará como decimal o en notación científica dependiendo de lo que sea mas corto.	double i=10.5; String d= String.format("Resultado: %g", i); System.out.println (d);	Resultado: 10.5000

- **Número** de caracteres que tendrá como mínimo y la precisión: **%[Ancho][.Precisión]Conversión**

```
String.format ("%5d",10);
String.format ("%5.3f",4.2f);
String np="Lavadora";
```

```
int u=10;
float ppu = 302.4f;
float p=u*ppu;
String output=String.format("Producto: %s; Unidades: %d; Precio por unidad: %.2f €;
Total: %.2f €", np, u, ppu, p);
System.out.println(output);
```

JAVA SWING

JOptionPane

- **showMessageDialog** : muestra un mensaje

  JOptionPane.showMessageDialog(null, "Hola mundo!!!");

- **showInputDialog**: nos permite introducir datos, lo devuelve como String:
  String ax = **JOptionPane.showInputDialog**("Teclee un numero: ");

- **showConfirmDialog**: nos permite responder si, no o cancelar:
  int ax = JOptionPane.**showConfirmDialog**(null, "Estas en java?");
  if(ax == JOptionPane.**YES_OPTION)**
  JOptionPane.showMessageDialog(null, "Has seleccionado SI.");
  else if(ax == JOptionPane.**NO_OPTION**)
  JOptionPane.showMessageDialog(null, "Has seleccionado NO.");

- **showOptionDialog**: selecciona una de las opciones que le indiquemos
  JOptionPane.**showOptionDialog**(parentComponent, message , title, optionType, messageType, icon, options, initialValue);

JTextField

- En el **JTextField** los eventos que más se utilizan son:
  - **ActionPerformed**: salta cuando se pulsa la tecla INTRO.
  - **Key** : **KeyPressed**, **KeyTypes**, **KeyRelease** (para manejar las teclas pulsadas y por ejemplo saber si son correctas o no)
  - **FocusGained**:salta cuando gana el foco
  - **FocusLost** :salta cuando ha perdido el foco

JButton

- Para capturar un evento, posicionados en el botón pulsamos el botón derecho seleccionamos **Events/Action/actionPerformed**

JComboBox

- El componente **JComboBox** permite seleccionar un elemento de una lista.
- Interface **ItemListener** (captura la selección de un ítem), contiene un método llamado **itemStateChanged**.
- Se puede capturar también con **ActionPerformed.**

Método	Significado
getSelectedItem()	Extrae el contenido del ítem seleccionado, devuelve un objeto.
getSelectedIndex()	Extrae el índice al que apunta el ítem seleccionado
addItem()	Inicializa los String que contendrá el **JComboBox**, debemos llamarlo tantas veces como elementos queramos cargar
getItemAt(int index)	Devuelve la lista de ítems al que apunta index
getItemCount()	Devuelve el nº de filas del combo
insertItemAt(Object nuevo,int index)	Inserta un objeto en el combo en la posición indicada por index
removeAllItems()	Elimina todos los ítems del combo

JTextArea

- El control de tipo **JTextArea** permite introducir múltiples líneas, a diferencia del control JTextField.

Seleccionar / Obtener Contenidos	
Método o constructor	**Función**
void **setText(String)** String **getText()**	Establece u obtiene el texto mostrado por el área de texto.
Implementar la Funcionalidad del área de texto	
Método	**Función**
void **selectAll()**	Selecciona todos los caracteres en el área de texto.
void **append(String)**	Añade el texto especificado al final de la zona de texto.
void **insert(String, int)**	Inserta el texto especificado en la línea indicada en int
void **replaceRange(String, int, int)**	Sustituye el texto entre las posiciones indicadas con la cadena especificada.

JComboBox

- El componente **JComboBox** permite seleccionar un elemento de una lista.
- Eventos que se pueden capturar:
  - **itemStateChanged**.
  - Se puede capturar también con **ActionPerformed.**

Método	Significado
getSelectedItem()	Extrae el contenido del ítem seleccionado, devuelve un objeto.
getSelectedIndex()	Extrae el índice al que apunta el ítem seleccionado
addItem()	Inicializa los String que contendrá el **JComboBox**, debemos llamarlo tantas veces como elementos queramos cargar
getItemAt(int index)	Devuelve la lista de ítems al que apunta index
getItemCount()	Devuelve el nº de filas del combo
insertItemAt(Object nuevo,int index)	Inserta un objeto en el combo en la posición indicada por index
removeAllItems()	Elimina todos los ítems del combo

JCheckBox

- El control **JCheckBox** permite implementar un cuadro de selección (básicamente un botón de dos estados)
- Evento que se pueden capturar:
  - **itemStateChanged**: cuando cambia el estado del item se ejecuta el método
  - **stateChage:** cuando cambia de estado de checkbox
  - boolean **isSelected():** nos devuelve true si el checkbox está seleccionado y false si no lo está.
- En **NetBeans** accedemos a las propiedades del jCheckBox, seleccionando **Event,** podemos seleccionar dentro de
- Esto creará un método llamado: **jCheckNombreStateChanged** y **jCheckNombreItemStateChanged**

RadioButton

- Los botones de opción **RadioButton** se utilizan para seleccionar solo un elemento de un conjunto de elementos.
- La diferencia principal de un CheckBox y un RadioButton radica principalmente en esta característica, ya que un conjunto de RadioButton solo puedes seleccionar un elemento, mientras que, en un conjunto de CheckBox puedes seleccionar uno, varios o todos los elementos.
- Otro control visual muy común es el **JRadioButton** que normalmente se muestran un conjunto de JRadioButton y permiten la selección de solo uno de ellos.
- Se les debe agrupar en un **ButtonGroup** para que actúen en conjunto, es decir cuando se selecciona uno automáticamente se deben deseleccionar los otros.
- Interfaz: **ChangeListener,** tiene el método **addChangeListener** (informa de qué objeto genera el evento) y el método **stateChanged** (se activa cuando cambia es estado de un radio)
- **Métodos:**
  - **isSeletd()**: nos devuelve true si el radio está seleccionado y false si no lo está
  - **setSelected**(true) : hace que el radio esté seleccionado.

JTable

- Un **JTable** es un componente visual de java que nos permite dibujar una tabla, de forma que en cada fila/columna de la tabla podamos poner el dato que queramos; un nombre, un apellido, una edad, un número, etc.
- **Constructores** que proporciona esta clase:
  - **JTable()**
  - **JTable(int numRows, int numColumns)**
  - **JTable(Object[][] rowData, Object[] columnNames)**
  - **JTable(TableModel dm)**
  - **JTable(TableModel dm, TableColumnModel cm)**
  - **JTable(TableModel dm, TableColumnModel cm, ListSelectionModelsm)**
  - **JTable(Vector rowData, Vector columnNames)**

- **El constructor: JTable(Object[][] rowData, Object[] columnNames)** nos permite construir una tabla a partir de dos parámetros; el primero de ellos: **rowData** es un array bidimencional de objetos que representa el **contenido de la tabla**, y el segundo: **columnNames** representa los **nombres de cada columna**, contenidos también en un array que por lo general es un array de String's.
- **Creando un Modelo de Tabla: La forma más rápida y sencilla**

  Con **DefaultTableModel** el constructor queda así:

  DefaultTableModel dtm= new DefaultTableModel(data, columnNames);

Después de haber creado el modelo de tabla, dtm en el ejemplo , se crea la tabla con el constructor correspondiente:

JTable table = new JTable(dtm);

- **Agregar una columna** Una vez hecho esto, cualquier modificación que se realice sobre el modelo de tabla se reflejará directamente en la tabla. Así, podemos agregar una columna:

  String[] newColumn= {"Flan","Pastel","Helado","Barquillo","Manzana" };

  dtm.addColumn("Postre",newColumn);

  Donde "Postre" es la columna de la cabecera **newColumn** es el contenido de dicha columna

- **Agregar una fila:**

  Object[] **newRow**={"Pepe", "Grillo","Tenis", new Integer(5),
  new boolean(false), "Pera"};

  dtm.addRow(newRow);

- **Modificar una celda en especial,** en este ejemplo la celda ubicada en la columna 1, fila 1:
  dtm.setValueAt("Catherine", 1, 1);

UTILIZACIÓN AVANZADA DE CLASES

- Se pueden distinguir diversos tipos de relaciones entre clases:
  - **Clientela**. Cuando una clase utiliza objetos de otra clase (por ejemplo, al pasarlos como parámetros a través de un método).
  - **Composición**. Cuando alguno de los atributos de una clase es un objeto de otra clase.
  - **Anidamiento**. Cuando se definen clases en el interior de otra clase.
  - **Herencia**. Cuando una clase comparte determinadas características con otra (clase base), añadiéndole alguna funcionalidad específica (especialización).

HERENCIA

- Declaración de una clase padre**:**

  [modificador] **class ClasePadre** { // Cuerpo de la clase ... }
- Declaración de una clase hija:

  [modificador] **class ClaseHija extends ClasePadre** { // Cuerpo de la clase ... }
- **Ejemplo:**

```
public class Persona {
    String nombre;
    String apellidos;
    GregorianCalendar fechaNacim; ...
}

public class Alumno extends Persona {
    String grupo;
    double notaMedia; ...
}
```

- No es posible acceder a miembros privados de una superclase.
- Los atributos heredados por una clase son, a efectos prácticos, iguales que aquellos que sean definidos específicamente en la nueva clase derivada.
- Del mismo modo que se heredan los atributos, también se heredan los métodos, convirtiéndose a partir de ese momento en otros métodos más de la clase derivada, junto a los que hayan sido definidos específicamente.
- Cuando sobrescribas un método heredado en Java puedes incluir la anotación **@Override**.
- La palabra reservada **super** es una referencia a la clase padre de la clase en la que te encuentres en cada momento (es algo similar a this, que representaba una referencia a la clase actual).
- De esta manera, podrías invocar a cualquier método de tu superclase (si es que se tiene acceso a él).

- De esta manera, el constructor de una clase derivada puede llamar primero al constructor de su superclase para que inicialice los atributos heredados y posteriormente se inicializarán los atributos específicos de la clase: los no heredados. Nuevamente, esta llamada también debe ser la primera sentencia de un constructor (con la única excepción de que exista una llamada a otro constructor de la clase mediante this).
- Si no se incluye una llamada a **super**() dentro del constructor, el compilador incluye automáticamente una llamada al constructor por defecto de clase base (llamada a super()).
- Esto da lugar a una llamada en cadena de constructores de superclase hasta llegar a la clase más alta de la jerarquía (que en Java es la clase Object).

CLASES ABSTRACTAS

- Son clases que nunca serán instanciadas, proporcionan un marco o modelo a seguir por sus clases derivadas dentro de una jerarquía de herencia.

  [modificador_acceso] **abstract class nombreClase** [herencia] [interfaces] { ... }

- Una clase puede contener en su interior **métodos** declarados como **abstract** (métodos para los cuales sólo se indica la cabecera, pero no se proporciona su implementación). En tal caso, la clase tendrá que ser también abstract. Esos métodos tendrán que ser posteriormente implementados en sus clases derivadas.
- Cuando trabajes con clases abstractas debes tener en cuenta:
  - Una clase abstracta sólo puede usarse para crear nuevas clases derivadas.
  - No se puede hacer un new de una clase abstracta. Se produciría un error de compilación.
  - Una clase abstracta puede contener métodos totalmente definidos (no abstractos) y métodos sin definir (métodos abstractos).

CLASES y MÉTODOS FINALES

- Una **clase** declarada como **final** no puede ser heredada, es decir, no puede tener clases derivadas.
- La jerarquía de clases a la que pertenece acaba en ella (no tendrá clases hijas):
  [modificador_acceso] final class nombreClase [herencia] [interfaces]
- Un **método** también puede ser declarado como **final**, en tal caso, ese método no podrá ser redefinido en una clase derivada:
  [modificador_acceso] final <tipo> <nombreMetodo> ([parámetros]) [excepciones]
- Si intentas redefinir un método final en una subclase se producirá un error de compilación.

INTERFACES

- Una **interfaz** en Java consiste esencialmente en una lista de declaraciones de métodos sin implementar, junto con un conjunto de constantes.
- Una clase puede implementar varias interfaces.
- Se utiliza la palabra reservada **interface** en lugar de class.
- Puede utilizarse el modificador public. Si incluye este modificador la interfaz debe tener el mismo nombre que el archivo .java en el que se encuentra (exactamente igual que sucedía con las clases). Si no se indica el modificador public, el acceso será por omisión o "de paquete" (como sucedía con las clases).
- Todos los atributos son de tipo **final** y **public** (tampoco es necesario especificarlo), es decir, constantes y públicos. Hay que darles un valor inicial.
- Todos los métodos son abstractos también de manera implícita (tampoco hay que indicarlo). No tienen cuerpo, tan solo la cabecera.

- Una **interfaz** consiste esencialmente en una lista de **atributos finales** (**constantes**) y **métodos abstractos** (**sin implementar**). Su sintaxis quedaría entonces:
  [public] **interface** <NombreInterfaz> {
  [public] [final] <tipo1> <atributo1>= <valor1>;
  [public] [final] <tipo2> <atributo2>= <valor2>;
  ...
  [public] [abstract] <tipo_devuelto1> <nombreMetodo1> ([lista_parámetros]);
  [public] [abstract] <tipo_devuelto2> <nombreMetodo2> ([lista_parámetros]);
  ...
  }

- Diferencia entre clases abstractas e interfaces:
  - Una clase no puede heredar de varias clases, aunque sean abstractas (herencia múltiple). Sin embargo, sí puede implementar una o varias interfaces y además seguir heredando de una clase.
  - Una interfaz no puede definir métodos (no implementa su contenido), tan solo los declara o enumera.
  - Una interfaz puede hacer que dos clases tengan un mismo comportamiento independientemente de sus ubicaciones en una determinada jerarquía de clases (no tienen que heredar las dos de una misma superclase, pues no siempre es posible según la naturaleza y propiedades de cada clase).
  - Una interfaz permite establecer un comportamiento de clase sin apenas dar detalles, pues esos detalles aún no son conocidos (dependerán del modo en que cada clase decida implementar la interfaz).
  - Las interfaces tienen su propia jerarquía, diferente e independiente de la jerarquía de clases.
- Dada una interfaz, cualquier clase puede especificar dicha interfaz mediante:
  class NombreClase implements NombreInterfaz {..}

CONECTAR JAVA CON ORACLE MEDIANTE JDBC

Pasos para conectar java con una base de datos:

1. JDBC: Cargar el driver:

   Class.forName("oracle.jdbc.driver.OracleDriver");

2. Establecer la conexión

   Connection conexion = DriverManager.getConnection
   ("jdbc:oracle:thin:@nombrepc:1521:NombreBD",“usuario",“clave");

3. Crear y Ejecutar sentencias SQL.

   Statement sentencia = conexion.createStatement();

   - Una vez creado el objeto **Statement** se pueden usar los siguientes métodos:
     - **executeQuery(String):** se utiliza para sentencias SQL que recuperan datos de un único objeto ResultSet, se utiliza para las sentencias **SELECT**.
     - **executeUpdate(String)**: se utiliza para sentencias que no devuelven un ResultSet como son las sentencias de manipulación de datos (DML): **INSERT, UPDATE y DELETE** y las sentencias de definición de datos(DDL): **CREATE, DROP y ALTER**. El método devuelve un entero indicando el número de filas que se vieron afectadas y en el caso de las sentencias DDL devuelve el valor 0.
     - **execute(String)**: se utiliza para sentencias que devuelven más de un ResultSet, se suele utilizar para ejecutar procedimientos almacenados.

- Para ejecutar la consulta los registros generados por la sentencia**:**

ResultSet resultado = sentencia.executeQuery("SELECT * FROM DEPARTAMENTOS_TAB");

4. Procesar resultado:
   - El resultado nos lo devuelve como un **ResultSet**, que es un objeto similar a una lista en la que está el resultado de la consulta.
   - Método **next**(): el puntero avanza al siguiente registro.
   - Los métodos **get** nos van devolviendo los valores de los campos de dicho registro.
   - Entre paréntesis se pone la posición de la columna en la SELECT o una cadena con el nombre de la columna en la tabla.
   - Por ejemplo, si el **primer** campo es DEPT_NO de tipo entero, se puede poner: resultado.**getInt(1)** o resultado.**getInt("DEPT_NO")**.

5. Liberar los recursos: **close()** (ResultSet, Statement, conexión)

Devolver resultado SQL: ResultSet

MÉTODO	DEVUELVEN OBJETOS DE TIPO
getString(columna)	String
getBoolean(columna)	Boolean
getByte(columna)	Byte
getShort(columna)	Short
getInt (columna)	Int
getLong(columna)	Long
getFloat(columna)	Float
getDouble(columna)	Double
getDate(columna)	Date
getTime(columna)	Time

Sentencias de manipulación de datos:

- **INSERT**:

```
String sql= "INSERT INTO DEPARTAMENTOS VALUES (" + dep +    ",'"+ dnombre + "','"+loc+"')";
Statement sentencia= conexion.createStatement();
int filas= sentencia.executeUpdate (sql); -- int devuelve el nº de filas insertadas
```

- **UPDATE:**

```
String sql= "UPDATE EMPLEADOS SET SALARIO=SALARIO + "+ subida+ " WHERE DEPT_NO=" +dept_no;
Statement sentencia = conexion.createStatement();
int filas=sentencia.executeUpdate(sql);
sentencia.close();
```

- **DELETE:**
  String sql = "**DELETE** EMPLEADOS WHERE IDDEPARTAMENTO =" + p_dept_no;
  Statement sentencia = conexion.**createStatement**();
  int filas = sentencia.**executeUpdate**(sql);

Sentencias preparadas con parámetros (PreparedStatement):

- Los métodos de **PreparedStatement** tienen los mismos nombres que en Statement **(executeQuery(), executeUpdate() y execute())** pero no se necesita enviar la cadena de caracteres con la orden SQL en la llamada ya que lo hace el método **prepareStatement(String)**.
  PreparedStatement sentencia = conexion.prepareStatement(sql);

- Cada marcador tiene un índice, el 1 correspondería a la primera ? que se encuentre en la cadena, el 2 a la segunda ? y así sucesivamente.

- METODOS:
  - **void setString(indice, String)**
  - **void setBoolean(indice, boolean)**
  - **void setByte(indice, byte)**
  - **void setShort(indice, short)**
  - **void setInt(indice, int)**
  - **void setLong(indice, long)**
  - **void setFloat(indice, float)**
  - **void setDouble(indice, double)**
  - **void setDate(indice, Date)**
  - **void setTime(indice, Time)**
  - **void setNull(indice, int tipoSQL)**

- **Select: Ejemplo** :
  String sql="**SELECT** APELLIDO,SALARIO FROM EMPLEADOS WHERE DEPT_NO=? AND OFICIO=?";
  PreparedStatement sentencia=conexion.**prepareStatement**(sql);
  sentencia.**setInt**(1,vdept_no);
  sentencia.**setString**(2,voficio);
  ResultSet resultado=sentencia.**executeQuery**();
  while(resul.next()){
  System.out.println("Nombre :"+resultado.**getString**(1)+" Salario :"+resultado.**getFloat**(2));

- **Inserción: Ejemplo**:

  String sql= "**INSERT** INTO DEPARTAMENTOS VALUES (?,?,?)";
  PreparedStatement sentencia= conexion.**prepareStatement**(sql);
  sentencia.**setInt**(1,vdept_no);
  sentencia.**setString**(2,vdnombre);
  sentencia.**setString**(3,vloc);
  int filas=sentencia.**executeUpdate**();

- **Actualización: Ejemplo**:

  String sql= "**UPDATE** EMPLEADOS SET SALARIO=SALARIO+? WHERE DEPT_NO= ?";
  PreparedStatement sentencia = conexion.**prepareStatement**(sql);

```
sentencia.setFloat(1,subida);
sentencia.setInt(2,vdept_no);
int filas=sentencia.executeUpdate();
```

- **Eliminación: Ejemplo**:

```
String sql= "DELETE EMPLEADOS  WHERE DEPT_NO= ?";
PreparedStatement sentencia = conexion.prepareStatement(sql);
sentencia.setInt(1,vdept_no);
int filas=sentencia.executeUpdate();
```

BASES DE DATOS OBJETO-RELACIONALES (BDOR)- ORACLE

TIPOS DE OBJETOS

- Un tipo de objeto representa una entidad del mundo real y se compone de los siguientes elementos:

```
CREATE TYPE DIRECCION_T AS OBJECT (
        CALLE           VARCHAR2(200),
        CIUDAD          VARCHAR2(200),
        PROV            CHAR(2),
        CODPOS          VARCHAR2(20)
);

CREATE TYPE CLIENTE_T AS OBJECT (
        CLINUM          NUMBER,
        CLINOMB         VARCHAR2(200),
        DIRECCION       DIRECCION_T,
        TELEFONO        VARCHAR2(20),
        FECHA_NAC       DATE,
        MEMBER FUNCTION EDAD RETURN NUMBER,
);
```

MÉTODOS

```
CREATE OR REPLACE TYPE BODY CLIENTE_T AS MEMBER
        FUNCTION EDAD RETURN NUMBER IS
        A NUMBER; D DATE;
        BEGIN
                D:= TODAY();
                A:= D.AÑO - FECHA_NAC.AÑO;
                IF (D.MES < FECHA_NAC.MES) OR ((D.MES = FECHA_NAC.MES)
                        AND (D.DIA < FECHA_NAC.DIA))        THEN
                         A:= A-1;
                END IF;
                RETURN A;
        END; -- FIN DE LA FUNCIÓN
END; -- FIN DEL BODY
```

CONSTRUCTORES

- Por ejemplo, las siguientes expresiones construyen dos objetos con todos sus valores.
  DIRECCION_T('AVENIDA SAGUNTO', 'PUZOL', 'VALENCIA', 'E-23523')
  CLIENTE_T(2347,'JUAN PÉREZ RUÍZ',**DIRECCION_T**('CALLE EO', 'ONDA', 'CASTELLÓN','34568'),'696-779789', 12/12/1981)

HERENCIA

- Para indicar que un tipo de objeto es heredado de otro hay que usar la palabra reservada **UNDER** y además hay que tener en cuenta que el tipo de objeto del que hereda debe tener la propiedad **NOT FINAL**
- La cláusula **OVERRIDING** se utiliza para redefinir el método.

```
CREATE TYPE PERSONA AS OBJECT(
    NOMBRE VARCHAR2(20),
    APELLIDOS VARCHAR2(30),
    MEMBER PROCEDURE VERDATOS
) NOT FINAL;

CREATE TYPE USUARIOPERSONA UNDER PERSONA (
    LOGIN VARCHAR(30),
    F_INGRESO DATE,
    CREDITO NUMBER,
    OVERRIDING MEMBER PROCEDURE VERDATOS
);
```

MÉTODOS DE COMPARACIÓN

- Para comparar los objetos de cierto tipo, es necesario indicar a Oracle cuál es el criterio de comparación. Para ello, hay que escoger entre un método MAP u ORDER, debiéndose definir al menos uno de estos métodos por cada tipo de objeto que necesite ser comparado. La diferencia entre ambos es la siguiente: **MAP y ORDER.**
- Un método **MAP** sirve para indicar cuál de los atributos del tipo se utilizará para ordenar los objetos del tipo, y por tanto se puede utilizar para comparar los objetos de ese tipo por medio de los operadores de comparación aritméticos (**<, >**). Por ejemplo, la siguiente declaración permite decir que los objetos del tipo **cliente_t** se van a comparar por su atributo clinum.

```
CREATE TYPE cliente_t AS OBJECT (
            clinum NUMBER,
            clinomb VARCHAR2(200),
            direccion direccion_t,
            telefono VARCHAR2(20),
            fecha_nac DATE,
    MAP MEMBER FUNCTION ret_value RETURN NUMBER,
    MEMBER FUNCTION edad RETURN NUMBER
);
CREATE OR REPLACE TYPE BODY cliente_t AS
    MAP MEMBER FUNCTION ret_value RETURN NUMBER IS BEGIN
                /*instrucciones a PL/SQL*/
                RETURN clinum – Se ordena por la columna clinum
    END;
END;
```

- Un método **ORDER** utiliza los atributos del objeto sobre el que se ejecuta para realizar un cálculo y compararlo con otro objeto del mismo tipo que toma como argumento de entrada. Este método devolverá un valor negativo si el parámetro de entrada es mayor que el atributo, un valor positivo si ocurre lo contrario y un cero si ambos son iguales. El siguiente ejemplo define un orden para el tipo **cliente_t** diferente al anterior. Sólo una de estas definiciones puede ser válida en un tiempo dado.

```
CREATE TYPE cliente_t AS OBJECT (
    clinum NUMBER,
    clinomb VARCHAR2(200),
    direccion direccion_t,
    telefono VARCHAR2(20),
    fecha_nac DATE,
    ORDER MEMBER FUNCTION cli_ordenados (x IN clientes_t) RETURN INTEGER,
    MEMBER FUNCTION edad RETURN NUMBER,
);

CREATE OR REPLACE TYPE BODY cliente_t AS
    ORDER MEMBER FUNCTION cli_ordenados (x IN cliente_t)        RETURN INTEGER IS
    BEGIN
            RETURN clinum - x.clinum; /*la resta de los dos números clinum*/
    END;
END;
```

TABLAS DE OBJETOS

```
CREATE TABLE clientes_año_tab OF cliente_t(clinum PRIMARY KEY);

CREATE TABLE clientes_antiguos_tab (
            año NUMBER,
            cliente cliente_t
    );
```

- La diferencia entre la primera y la segunda tabla es que la primera almacena objetos con su propia identidad (OID) y la segunda no es una tabla de objetos, sino una tabla con una columna con un tipo de datos de objeto. Es decir, la segunda tabla tiene una columna con un tipo de datos complejo pero sin identidad de objeto.
- Por ejemplo, se puede ejecutar una de las dos instrucciones siguientes.
- En esta instrucción instrucción, la tabla clientes_año_tab se considera como una tabla con varias columnas cuyos valores son los especificados.

```
INSERT INTO clientes_año_tab VALUES( 347,'Juan Pérez Ruíz',
direccion_t('Calle Castalia','Onda','Castellón','34568'), '696-779789',12/12/1981);
```

- En el segundo caso, es una tabla de objetos que en cada fila almacena un objeto. En esta instrucción la cláusula VALUE permite visualizar el valor de un objeto.

```
SELECT VALUE(c) FROM clientes_año_tab c WHERE c.clinomb = 'Juan Pérez Ruíz';
```

- Las reglas de integridad, de clave primaria, y el resto de propiedades que se definan sobre una tabla, sólo afectan a los objetos de esa tabla, es decir no se refieren a todos los objetos del tipo asignado a la tabla.

REFERENCIA ENTRE OBJETOS

- Un atributo de tipo REF almacena una referencia a un objeto del tipo definido, e implementa una relación de asociación entre los dos tipos de objetos. Estas referencias se pueden utilizar para acceder a los objetos referenciados y para modificarlos; sin embargo, no es posible operar sobre ellas directamente. Para asignar o actualizar una referencia se debe utilizar siempre REF o NULL.

- El siguiente ejemplo define un atributo de tipo REF y restringe su dominio a los objetos de cierta tabla.

```
CREATE TABLE clientes_tab OF cliente_t;

CREATE TYPE ordenes_t AS OBJECT (
        ordnum NUMBER,
        cliente REF clientes_t,
        fechpedido DATE,
        direcentrega direccion_t
);

CREATE TABLE ordenes_tab OF ordenes_t (
        PRIMARY KEY (ordnum),
        SCOPE FOR (cliente) IS clientes_tab
);
```

- Cuando se borran objetos de la BD, puede ocurrir que otros objetos que referencien a los borrados queden en estado inconsistente. Estas referencias se denominan dangling references, y Oracle proporciona el predicado llamado IS DANGLING que permite comprobar cuándo sucede esto.

TIPOS DE DATOS COLECCIÓN

- Para poder implementar relaciones 1:N, Oracle permite definir tipos colección. Un dato de tipo colección está formado por un número indefinido de elementos, todos del mismo tipo. De esta manera, es posible almacenar en un atributo un conjunto de tuplas en forma de array (VARRAY), o en forma de tabla anidada.
- Al igual que los tipo objeto, los tipo colección también tienen por defecto unas funciones constructoras de colecciones cuyo nombre coincide con el del tipo. Los argumentos de entrada de estas funciones son el conjunto de elementos que forman la colección separados por comas y entre paréntesis, y el resultado es un valor del tipo colección.
- En Oracle es posible diferenciar entre un valor nulo y una colección vacía. Para construir una colección sin elementos se puede utilizar la función constructora del tipo seguida por dos paréntesis sin elementos dentro.

EL TIPO VARRAY

- Un array es un conjunto ordenado de elementos del mismo tipo. Cada elemento tiene asociado un índice que indica su posición dentro del array. Oracle permite que los VARRAY sean de longitud variable, aunque es necesario especificar un tamaño máximo cuando se declara el tipo VARRAY. Las siguientes declaraciones crean un tipo para una lista ordenada de precios, y un valor para dicho tipo.

```
CREATE TYPE precios AS VARRAY(10) OF NUMBER(12); precios('35', '342', '3970');
```

- Se puede utilizar el tipo VARRAY para:
  - Definir el tipo de dato de una columna de una tabla relacional.
  - Definir el tipo de dato de un atributo de un tipo de objeto.
  - Para definir una variable PL/SQL, un parámetro, o el tipo que devuelve una función.

- Cuando se declara un tipo VARRAY no se produce ninguna reserva de espacio. Si el espacio que requiere lo permite, se almacena junto con el resto de columnas de su tabla, pero si es demasiado largo (más de 4000 bytes) se almacena aparte de la tabla como un BLOB.

```
CREATE TYPE lista_tel_t AS VARRAY(10) OF VARCHAR2(20) ;
CREATE TYPE cliente_t AS OBJECT (
        clinum NUMBER,
        clinomb VARCHAR2(200),
        direccion direccion_t,
        lista_tel lista_tel_t
);
```

TABLAS ANIDADAS

- Una tabla anidada es un conjunto de elementos del mismo tipo sin ningún orden predefinido. Estas tablas solamente pueden tener una columna que puede ser de un tipo de datos básico de Oracle, o de un tipo de objeto definido por el usuario. En este último caso, la tabla anidada también puede ser considerada como una tabla con tantas columnas como atributos tenga el tipo de objeto.
- El siguiente ejemplo declara una tabla que después será anidada en el tipo ordenes_t. Los pasos de todo el diseño son los siguientes:

1. Se define el tipo de objeto **linea_t** para las filas de la tabla anidada.

```
CREATE TYPE linea_t AS OBJECT (
    linum NUMBER,
    item VARCHAR2(30),
    cantidad NUMBER,
    descuento NUMBER(6,2)
);
```

2. Se define el tipo colección tabla **lineas_pedido_t** para después anidarla.

```
CREATE TYPE lineas_pedido_t AS TABLE OF linea_t ;
```

Esta definición permite utilizar el tipo colección **lineas_pedido_t** para:

- Definir el tipo de dato de una columna de una tabla relacional.
- Definir el tipo de dato de un atributo de un tipo de objetos.
- Para definir una variable PL/SQL, un parámetro, o el tipo que devuelve una función.
-

3. Se define el tipo objeto **ordenes_t** y su atributo **pedido** almacena una tabla anidada del tipo **lineas_pedido_t**.

```
CREATE TYPE ordenes_t AS OBJECT (
    ordnum NUMBER,
    cliente REF cliente_t,
    fechpedido DATE,
    fechentrega DATE,
    pedido lineas_pedido_t,
    direcentrega direccion_t) ;
```

4. Se define la tabla de objetos **ordenes_tab** y se especifica la tabla anidada del tipo **lineas_pedido_t.**

```
CREATE TABLE ordenes_tab OF ordenes_t (
    ordnum PRIMARY KEY,
    SCOPE FOR (cliente) IS clientes_tab)
    NESTED TABLE pedido STORE AS pedidos_tab) ;
```

- Este último paso es necesario realizarlo porque la declaración de una tabla anidada no reserva ningún espacio para su almacenamiento. Lo que se hace es indicar en qué tabla (pedidos_tab) se deben almacenar todas las líneas de pedido que se representen en el atributo pedido de cualquier objeto de la tabla ordenes_tab. Es decir, todas las líneas de pedido de todas las ordenes se almacenan externamente a la tabla de ordenes, en otra tabla especial. Para relacionar las tuplas de una tabla anidada con la tupla a la que pertenecen, se utiliza una columna oculta que aparece en la tabla anidada por defecto. Todas las tuplas de una tabla anidada que pertenecen a la misma tupla tienen el mismo valor en esta columna (NESTED_TABLE_ID).

- A diferencia de los VARRAY, los elementos de las tablas anidadas (NESTED_TABLE) sí pueden ser accedidos individualmente, y es posible poner condiciones de recuperación sobre ellos. En la próxima sección veremos, una forma conveniente de acceder individualmente a los elementos de una tabla anidada mediante un cursor anidado. Además, las tablas anidadas pueden estar indexadas.

INSERCIÓN Y ACCESO A LOS DATOS

ALIAS

- En una base de datos con tipos y objetos, lo más recomendable es utilizar siempre alias para los nombres de las tablas. El alias de una tabla debe ser único en el contexto de la consulta. Los alias sirven para acceder al contenido de la tabla, pero hay que saber utilizarlos adecuadamente en las tablas que almacenan objetos.
- El siguiente ejemplo ilustra cómo se deben utilizar.

```
CREATE TYPE persona AS OBJECT (nombre VARCHAR(20));
CREATE TABLE ptab1 OF persona;
CREATE TABLE ptab2 (c1 persona);
CREATE TABLE ptab3 (c1 REF persona);
```

- La diferencia entre las dos primeras tablas está en que la primera almacena objetos del tipo persona, mientras que la segunda tabla tiene una columna donde se almacenan valores del tipo persona.
- Considerando ahora las siguientes consultas, se ve cómo se puede acceder a estas tablas.

```
SELECT nombre FROM ptab1;          Correcto
SELECT c1.nombre FROM ptab2;       Incorrecto
SELECT p.c1.nombre FROM ptab2 p;   Correcto
SELECT p.c1.nombre FROM ptab3 p;   Correcto
SELECT p.nombre FROM ptab3 p;      Incorrecto
```

INSERCIÓN DE REFERENCIAS

- La inserción de objetos con referencias implica la utilización del operador REF para poder insertar la referencia en el atributo adecuado. La siguiente sentencia inserta una orden de pedido en la tabla definida anteriormente.

```
INSERT INTO ordenes_tab
    SELECT 3001, REF(C),'30-MAY-1999', NULL      --se seleccionan los valores de los 4 atributos de la tabla
    FROM cliente_tab C WHERE C.clinum= 3;
```

- El acceso a un objeto desde una referencia REF requiere primero referenciar al objeto. Para realizar esta operación, Oracle proporciona el operador DEREF. No obstante, utilizando la notación de punto también se consigue referenciar a un objeto de forma implícita.
- Observemos el siguiente ejemplo:

```
CREATE TYPE persona_t AS OBJECT (
    nombre VARCHAR2(30),
    jefe REF persona_t ) ;
```

- Si x es una variable que representa a un objeto de tipo persona_t, entonces las dos expresiones siguientes son equivalentes:

```
x.jefe.nombre
y.nombre,
y=DEREF(x.jefe)
```

- Para obtener una referencia a un objeto de una tabla de objetos, se puede aplicar el operador REF como muestra el siguiente ejemplo:

```
CREATE TABLE persona_tab OF persona_t;
DECLARE ref_persona REF persona_t;
SELECT REF(pe) INTO ref_persona
        FROM persona_tab pe WHERE pe.nombre= 'Juan Pérez Ruíz';
```

- Simétricamente, para recuperar un objeto desde una referencia es necesario usar DEREF, como muestra el siguiente ejemplo que visualiza los datos del jefe de la persona indicada:

```
SELECT DEREF(pe.jefe)
FROM persona_tab pe WHERE pe.nombre= 'Juan Pérez Ruíz';
```

LLAMADAS A MÉTODOS

- Para invocar un método hay que utilizar su nombre y unos paréntesis que encierren sus argumentos de entrada. Si el método no tiene argumentos, se especifican los paréntesis aunque estén vacíos. Por ejemplo, si tb es una tabla con la columna c de tipo de objeto t, y t tiene un método m sin argumentos de entrada, la siguiente consulta es correcta:

```
SELECT p.c.m( ) FROM tb p;
```

INSERCIÓN EN TABLAS ANIDADAS

- Además del constructor del tipo de colección disponible por defecto, la inserción de elementos dentro de una tabla anidada puede hacerse siguiendo estas dos etapas:
  1. Crear el objeto con la tabla anidada y dejar vacío el campo que contiene las tuplas anidadas.
  2. Comenzar a insertar tuplas en la columna correspondiente de la tupla seleccionada por una subconsulta. Para ello, se tiene que utilizar la palabra clave **THE** con la siguiente sintaxis:

```
INSERT INTO THE (subconsulta) (tuplas a insertar)
```

- Esta técnica es especialmente útil si dentro de una tabla anidada se guardan referencias a otros objetos. El siguiente ejemplo ilustra la manera de realizar estas operaciones sobre la tabla de ordenes (ordenes_tab) definida anteriormente:

```
INSERT INTO ordenes_tab          --inserta una orden
        SELECT 3001, REF(C), SYSDATE,'30-MAY-1999',lineas_pedido_t(),NULL
        FROM cliente_tab C WHERE C.clinum= 3 ;

INSERT INTO --selecciona el atributo pedido de la orden
THE ( SELECT P.pedido FROM ordenes_tab P WHERE P.ordnum = 3001)
        VALUES (linea_t(30, NULL, 18, 30)); --inserta una linea de pedido anidada
```

- Para poner condiciones a las tuplas de una tabla anidada, se pueden utilizar cursores dentro de un SELECT o desde un programa PL/SQL. Veamos aquí un ejemplo de acceso con cursores. Utilizando el ejemplo de la sección anterior, vamos a recuperar el número de las ordenes, sus fechas de pedido y las líneas de pedido que se refieran al item 'CH4P3'.

```
SELECT ord.ordnum, ord.fechpedido,
        CURSOR (SELECT * FROM TABLE(ord.pedido) lp WHERE lp.item= 'CH4P3')
FROM ordenes_tab ord;
```

- La cláusula THE también sirve para seleccionar las tuplas de una tabla anidada. La sintaxis es como sigue:

```
SELECT ... FROM THE (subconsulta) WHERE ...
```

- Por ejemplo, para seleccionar las primeras dos líneas de pedido de la orden 8778 se hace:

```
SELECT lp FROM THE
(SELECT ord.pedido FROM ordenes_tab ord WHERE ord.ordnum= 8778) lp
WHERE lp.linum<3;
```